Programverktøy og algoritmer for automatisk utlegg av transistorer

Dag Asheim

15. mai 1995

Forord

Dette er en hovedoppgave i databehandling til cand. scient. graden. I en slik oppgave er det mange som har kommet med små og store innspill. Her skal jeg bare trekke frem de som har betydd mest.

Først vil jeg rette en takk til min veileder Kjell Øystein Arisland som også har kommet med mange av grunnideen bak oppgaven. Han har tålmodig lært meg mye grunnleggende digitalteknikk, og fått loset meg igjennom oppgaven. Takk til Sigbjørn Næss som vekket min interesse for algoritmer innenfor VLSI og som foreslo at jeg skulle kontakte Kjell Øystein Arisland i første omgang.

Takk til Terje Knudsen som har svart på mange spørsmål om hvordan kretser konstrueres, og som dessuten har lest oppgaven og kommet med mange verdifulle kommentarer. Olav Asheim har lest oppgaven og foreslått forbedringer som har gjort fremstillingen bedre. Dette har også Arne Kristian Groven gjort, men han var i tillegg sterkt inne i den avgjørende sluttfasen (les: Han sov ikke mye natt til 15. mai.).

Noen som indirekte har gjort oppgaven bedre, er Gerd Gran Andreasen og Mathilde Asheim som har stilt opp som barnevakter, og dermed gitt meg mulighet til å arbeide så mye med oppgaven som jeg har gjort. En generell takk også til medstudenter og andre på Ifi som har gjort studiene til en positiv opplevelse.

Men den største takken går til Kari Asheim som både har gitt meg muligheten til å arbeide intensivt med oppgaven, og som samtidig har gitt meg uvurderlig faglig hjelp helt til det aller siste.

Alle feilene som gjenstår kan jeg takke meg selv for.

Dag.

Innhold

Kapittel 1

Innledning

Det som har skjedd på datafronten etter krigen er en av de største revolusjoner innen menneskets teknologiske utvikling. Det kan sies å være på linje med den industrielle revolusjon i betydning.

1.1 Miniatyrisering

Mens man tidligere tenkte på datamaskiner som elektroniske kjempehjerner som trengte store rom for å få plass, reagerer ingen lenger på at vaskemaskinen styres av en liten mikroprosessor.

Mye av denne utviklingen kommer av at man har funnet bedre teknikker. I stedet for å ha store radiorør som grunnelement, gikk man over til transistorer, og siden integrerte kretser. I begynnelsen kunne man bare legge noen få transistorer inn på en brikke (eng. *chip*), men nå blir brikker med millioner av transistorer masseprodusert.

1.1.1 Grunnene for å miniatyrisere

At brikkene stadig blir mindre er ikke bare en kuriositet som gjør at størrelsen på lommekalkulatorene kan minskes.

- For det første er det avgjørende for hvor raske prosessorene kan være at transistorene ligger tett. Siden det alltid vil være et sterkt ønske om stadig raskere prosessorer er dette viktig.
- Vi ønsker å holde funksjonen på en brikke. Det blir raskere og billigere enn hvis den må fordeles over flere brikker.
- Det er sterke økonomiske hensyn som gjør at det er ønskelig med små brikker. I avsnitt 1.2.11 forklares det nærmere hvorfor det er slik.

1.1.2 Hvordan lage enda mindre brikker?

For å lage enda mindre brikker kan man forbedre den fysiske fremstillingsteknikken (enda mer) slik at lederne kan ligge tettere og transistorene kan være mindre. En annen viktig kilde til å lage mindre brikker er å utnytte det tilgjengelige arealet bedre, ved å legge transistorer og ledere på en mer hensiktsmessig måte. Det å lage gode utlegg er nemlig vanskelig og tidkrevende.

Et naturlig spørsmål vil være om ikke en datamaskin kan gjøre det for oss. De siste 15–20 årene har det blitt laget slike programmer, men hittil har det vist seg at en menneskelig ekspert klarer å lage mer kompakte utlegg enn de beste automatiske verktøyene. Dette tyder på at det er mer å hente på å gjøre det maskinelt, og pga. den økonomiske betydningen er dette et felt hvor det forskes mye.

1.1.3 Mål for oppgaven

Målet for denne oppgaven er å bidra til utvikling av automatiske verktøy og algoritmer som skal oppfylle følgende spesifikasjon:

Gitt: En liste over hvilke transistorer som finnes, og hvordan de skal kobles sammen.

Ønsket resultat: Et ferdig utlegg, med eksakt plassering av transistorer og ledere, slik at totalt areal blir så lite som mulig, samtidig som designreglene overholdes.

Verktøyet, som vi kan kalle en modulgenerator, er bare ment å fungere for rundt 200 transistorer. Hvis man har flere transistorer enn det, er det naturlig å kombinere dette verktøyet med et annet som er egnet for å sette sammen moduler.

1.2 Fabrikasjon av integrerte kretser

For å skjønne denne oppgaven må man vite litt om hvordan en brikke blir til. Dette er forklart grundigere i f.eks. (Lundh, Søråsen, Bayegan, & Pedersen, 1983; Weste & Eshraghian, 1985) men jeg tar allikevel med det viktigste som er aktuelt for denne oppgaven. Oppgaven begrenser seg til den vanligste VLSI fabrikasjonsteknikken, CMOS — Complimentary Metal-Oxide Semiconductor.

1.2.1 Silisium-skive

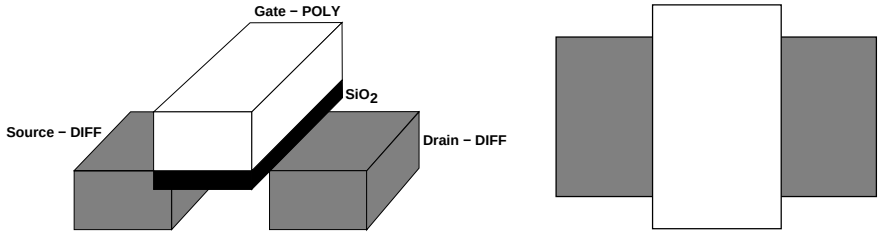
Produksjonsprosessen starter med at man skal lage en tynn skive av metallet silisium. For at skiven skal være egnet til bruk i VLSI må silisumet ha krystalinsk form. For å lage den, tar man en liten bit med silisiumskrystall, og lar den rotere i en smeltedigel med silisium oppvarmet til smeltepunktet. Ved langsom avkjøling vil krystallet i midten fungere som en katalysator og sørge for at alt silisiumet går over i krystalinsk form. Denne prosessen kalles gjerne "å gro" krystall. Hvis dette går bra har man en sylinder som man så kan dele opp i skiver. På overflaten av disse skivene blir de forskjellige komponentene laget. En slik skive (eng. *wafer*) gir grunnlaget for flere brikker.

1.2.2 Halvleder

Poenget med å bruke silisium er at det er en halvleder (eng. *semiconductor*), dvs. at ledningsevnen ligger mellom strømførende og strømisolerende. Ledningsevnen blir sterkt endret hvis det ligger "urene" atomer i krystallstrukturen. Å tilføre "urene" atomer kalles doping. Det kan enten tilføres atomer som har et overskudd med frie elektroner (området blir strømførende), eller et underskudd på elektroner (området blir strømisolerende). Silisium i et strømførende område sier vi er av N-typen, mens strømisolerende silisium er av P-typen. Det interessante skjer når vi kobler sammen N-type og P-type silisium.

1.2.3 Transistor

Dopet silisium kalles diffusjon (som blir forkortet til "diff"). Over skiven legges det på et strømisolerende oksyd-lag (SiO_2). Deretter legger man på polykrystallinsk silisium (som blir forkortet til polysilisium eller bare "poly"). Dersom poly overlapper diffusjon og oksyd-laget etter bestemte regler, vil det bare gå strøm i diffusjonen hvis det også er strøm i poly-laget. Dermed har vi en transistor (se figur 1.1).



(a) Sett fra siden (b) Sett ovenfra

Figur 1.1: En transistor. Diffusjon (grått) nederst, polysilisium (hvitt) øverst og SiO_2 (sort) i midten. Underlaget er silisium-substrat, og vil bare lede strøm mellom source og drain hvis det er strøm i tilhørende gate.

Source, drain og gate er navnene på de tre tilkoblingpunktene, terminalene, som en transistor har. De engelske termene er så innarbeidet at jeg ikke vil prøve å oversette dem.

1.2.4 Brønner og to typer transistorer

I CMOS er det to typer transistorer, N og P. En N-transistor virker slik som beskrevet over, mens en P-transistor bare lar det gå strøm mellom source og drain dersom det *ikke* går strøm på gate-terminalen. For å lage to typer transistorer på denne måten, lar man den ene typen ligge i en vanlig del av underlaget (substratet), mens den andre ligger i en brønn (eng. *well*). Hvis det er P-transistorene som ligger i brønner, vil brønnene være av N-typen, og omvendt. Om vi har N-brønner eller P-brønner er avhengig av produksjonsprosessen som brukes — vi kan også ha begge deler. Denne oppgaven blir ikke berørt disse detaljene. Hovedpoenget er å skille mellom de to typene transistorer og to typer underlag.

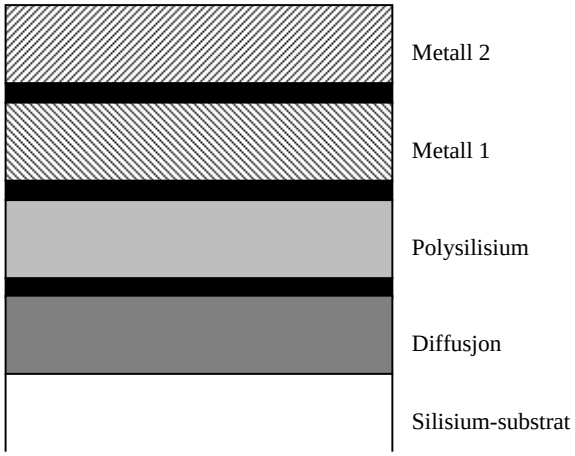
For at en brønn skal ha de riktige elektriske egenskapene, må den ha en eller flere brønnkontakter ned i underlaget. Andre komplikasjoner med brønner er at de må ha en viss størrelse, og at det ikke kan ligge transistorer for nær kanten av en brønn. Derfor er det vanlig å samle flere transistorer av samme type i en litt større brønn, slik at de kan dele på den ekstra plassen til brønnkanter og -kontakter. Man trenger omtrent en brønnkontakt per 5–10 transistorer.

1.2.5 Metall-lag

Hvis to ledere skal krysse hverandre (uten at de blir elektrisk sammenkoblet), må de gå i forskjellige lag. Fordi det er forbindelsen mellom transistorene som er den største begrensning mot å få tettere utlegg, går utviklingen i retning av flere lag til å legge ledere i.

Over diffusjon og poly-lagene, legges det derfor på alternerende lag med isolerende oksyd og med elektrisk ledende metall. Alle ledere over en viss lengde eller der det stilles store krav til hastighet/ledningsevne, f.eks strømforsyning og klokkesignaler, bør ledes i metall-lag.

Figur 1.2 illustrerer de forskjellige lagene i en fabrikasjonsprosess med to metall-lag.



Figur 1.2: Lagene i en 2-metalls CMOS-prosess sett fra siden. Substrat og diffusjon er begge en del av den opprinnelige skiven (buried layers), mens lagene over er lagt til etterpå (grown layers).

1.2.6 Via-kontakter

For å koble en leder som går i et lag, sammen med en leder på et annet lag, bruker man det som kalles en via, en via-forbindelse eller via-kontakt. En via lages ved å unnlate å legge isolerende oksyd mellom de to lagene som skal forbindes.

Selv om en via skaper kontakt mellom lagene, er den elektriske forbindelsen ikke like god som over en vanlig leder. Den skaper uønskede elektriske parasitteffekter, som går ut over ytelsen til kretsen. På grunn av dette, og fordi viaer øker sannsynligheten for feil ved produksjonen, er det viktig å ikke ha flere viaer enn man trenger.

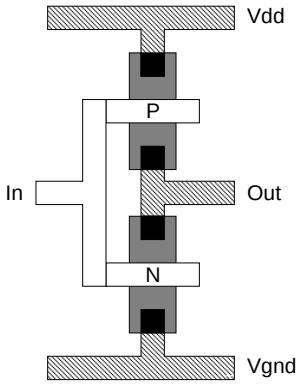
1.2.7 Manhattan-geometri

Det er vanlig å begrense plasseringen av komponentene og lederne til å ligge parallelt med sidekantene (som har $90°$ vinkel mellom seg). Dette kalles gjerne Manhattan-geometri, pga. likheten med gatebildet i Manhattan, New York. Dette er hovedsaklig motivert utifra en kombinasjon av å gjøre designet av utlegg enklere og å gjøre den fysiske fabrikasjonen enklere.

Manhattan-avstanden mellom to punkter er summen av den horisontale og vertikale avstanden.

1.2.8 Fra transistorer til krets

Det som gjør transistorene så interessante, er at man ved å sette sammen transistorene på forskjellige måter kan bygge opp avanserte logiske funksjoner. Vi kan f.eks sette sammen to transistorer slik som i figur 1.3.



Figur 1.3: En inverter i CMOS.

V_{dd} er forsyningsspenningen, mens V_{gnd} er jord. Brønnen(e) er ikke tegnet inn, men de to transistorene skal ligge i forkjellige typer brønner/underlag, som nevnt i avsnitt 1.2.4.

Hvis det er strøm på In, vil P-transistoren ikke lede, mens N-transistoren vil. Derfor blir Out trukket ned til jord. Hvis det ikke er strøm på In, vil Out bli trukket opp til V_{dd}. Denne lille kretsen beregner altså den logiske funksjonen NOT. Mer avanserte funksjoner kan trenge klokkesignaler og tilbakekobling (feedback) for å fungere, men dette er hovedprinsippene.

1.2.9 Nettliste

Det hender at en leder går mellom mer enn to endepunkter, slik som lederen som er koblet til In og de to gate-terminalene i forrige eksempel. Et annet ord på en leder er nett. En liste over hvilke nett som finnes, og hvordan de er koblet til transistorene, kalles en nettliste.

En nettliste sier ikke noe om plasseringen av transistorer og ledere, så en nettliste kan realiseres som et utlegg på mange forskjellige måter. Utfordringen er å finne et utlegg som bruker så lite plass som mulig.

1.2.10 Designregler

Ved fabrikasjon av VLSI kretser, opererer man ofte på grensen av det man får til. Ved en fabrikk kan man kanskje lage litt mindre transistorer enn det som går på en annen fabrikk. Den andre fabrikken tilbyr kanskje at transistorene kan ligge tettere. Samtidig endrer disse faktorene seg fort, etter som fabrikkene utvikler fabrikasjonsteknologien videre.

For at de som lager brikker skal vite hva de skal holde seg til, har alle leverandørene et sett med såkalte designregler (eng. *design rules*) som beskriver toleransegrenser som varierer mellom forskjellige fabrikasjonsprosesser. Hvis kretsen oppfyller kravene fra designreglene er det gode sjanser for at den vil virke.

Designreglene spesifiserer ting som:

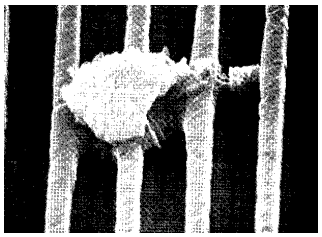
- Linjebredde — hvor tynne ledere/transistorer kan vi lage
- Hvilke lag som er tilstede
- Geometriske regler — hvor tett man kan legge forskjellige komponenter uten at det blir kontakt, osv.

- Elektriske prosess-avhengige regler — beskriver de elektriske egenskapene som vil avgjøre om man har for mange viaer osv.

Isteden for å si at avstanden mellom to transistorer må være minimum 1μm, kan man bruke en parameter λ. På den måten er det lett å skalere hele utlegget til å bruke en annen verdi for λ.

1.2.11 Utbytte

Det vil alltid være noen feil spredt utover en silisium-skive, enten fordi krystallstrukturen som dannes er ujevn eller pga. feil under produksjonen (se figur 1.4).



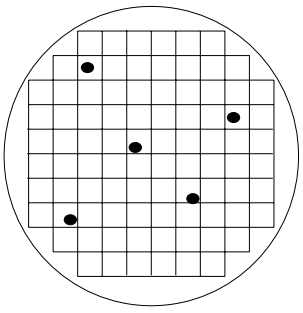
Figur 1.4: En ansamling av metall som feilaktig kortslutter ledere. Figuren er hentet fra (Doy et al., 1987).

Disse feilene fører til at det er mye billigere å lage små brikker enn store. Grunnen til at feilene slår ut så mye i små kretsers favør, har med begrepet utbytte (eng. *yield*) å gjøre. Utbyttet for en gitt kretsproduksjon er definert som

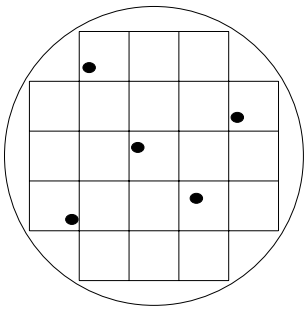
$$\text{Utbytte} = \frac{\text{antall vellykkede kretser}}{\text{antall mulige kretser}},$$

eller alternativt sannsynligheten for at en krets blir vellykket. Utbyttet er i stor grad styrt av kvalitetssikring under produksjonen (lokalene må være støvfrie o.l.).

Kretsdesigneren må selvsagt følge designreglene for å sikre at utbyttet blir så høyt som mulig. Men også ved å konstruere små brikker, bidrar designeren til å øke utbyttet. Grunnen er at utbyttet sannsynligvis blir mye lavere hvis man skal lage store kretser, enn hvis man skal lage små. Dette illustreres best ved figur 1.5.



(a) Små kretser. Utbyttet blir $\frac{83}{88} \approx 94.3\%$.



(b) Store kretser. Utbyttet blir $\frac{16}{21} \approx 76.2\%$.

Figur 1.5: Feil og utbytte. Fysiske feil er markert med sorte prikker. Figuren er basert på en tilsvarende fra (Riezenman, 1991).

I figur 1.5 (a) ser vi en skive som skal brukes til å produsere små kretser. Optimalt kan vi få 88 ferdige kretser, men på grunn av fysiske feil (markert med sorte prikker) må noen av kretsene kasseres slik at vi sitter igjen med 83 brukbare kretser. Utbyttet blir $\frac{83}{88} \approx 94.3\%$.

I 1.5 (b) har vi en tilsvarende situasjon. Forskjellen er at vi ønsker å lage litt større kretser. Vi ser at med samme feildistribusjon blir utbyttet nå bare $\frac{16}{21} \approx 76.2\%$. Hvis vi vil lage en kjempe-krets er sjansen for at den ikke skal ha noen feil være nærmest lik 0.

Ofte har man kretser hvor størrelsen er på grensen av det som kan produseres med rimelig godt utbytte. Hvis man klarer å presse kretsen inn på et litt mindre areal vil man straks få langt flere vellykkede kretser ut. Hvis man får dobbelt så mange vellykkede kretser ut, vil den totale kostanden for å prosessere et bestemt antall kretser halveres. Her er det snakk om potensielt svært høye beløp, og dermed tilsvarende store potensielle innsparinger.

1.3 Kompleksitet på problemer

Under prosessen med å la datamaskiner konstruere integrerte kretser kommer man opp i problemer som man gjerne skulle ha en optimal, eller nær optimal løsning på. Samtidig er mange av disse problemene svært vanskelige å finne gode løsninger på.

1.3.1 NP-kompletthet

NP-kompletthet (Garey & Johnson, 1979) er et begrep fra kompleksitetsteori som har fått stor utbredelse i de siste årene. Grunnen er at svært mange interessante problemer er NP-komplette, blant annet mange av problemene som blir behandlet i denne oppgaven. Dessverre virker det som om ikke alle helt har skjønt hva dette begrepet innebærer, og bruker det omtrent som et synonym for "vanskelig". En slik forenklig fører til at man går glipp av noen nyanser.

Noen sier at NP-kompletthet betyr at et problem ikke kan løses optimalt i polynomisk tid (f.eks orden $O(n^2)$, $O(n^3 \ln n)$ eller $O(n^4)$) i motsetning til eksponentiell tid (f.eks orden $O(2^n)$ eller $O(n!)$). De fleste forskere tror at dette er riktig, men ingen har klart å konstruere noe bevis, eller motbevis, for dette. Uansett om man senere finner at man kan løse NP-komplette problemer i polynomisk tid, vil sannsynligvis algoritmene ha en orden som gjør en optimal løsning uaktuell i mange sammenhenger.

Et annet punkt som fordrer presisjon, er at man skiller mellom desisjonsproblemer (problemer hvor svaret enten er "ja" eller "nei") og søkeproblemer (problemer hvor svaret kan være f.eks "nei", "45", eller en beskrivelse av hvordan man skal plassere komponentene slik at et VLSI utlegg blir minst mulig). Slik NP-kompletthet er definert, er det bare desisjonsproblemer som kan være NP-komplette. Siden de fleste interessante problemer er søkeproblemer ("Finn den minste k som oppfyller...!"), skulle man kanskje tro at NP-kompletthet ikke var anvendbart i mange tilfeller. I virkeligheten kan de fleste problemer formuleres som desisjonsproblemer ("Går det an å finne en k som er mindre enn k', og som oppfyller...?"). Men det betyr at man må være presis når man snakker om dette. Derfor vil jeg mange steder skrive "NP-komplett gitt en passende formulering".

1.3.2 Hvordan håndtere kompleksitet?

For små instanser vil det ikke være av så stor betydning at algoritmene er av eksponentiell orden, men man skal ikke opp i spesielt store instanser før optimale løsninger i rimelig tid er umulig.

En mulighet er å analysere problemet grundigere. Det er ofte små ting som skiller et NP-komplett problem fra et som kan løses i polynomisk tid. For eksempel vil mange NP-komplette grafproblemer bli enkle å løse dersom vi begrenser oss til grafer hvor hver node bare har to tilhørende kanter. Hvis vi faktisk bare ønsker å løse problemet for slike grafer, behøver vi ikke bekymre oss om at det generelle tilfellet er NP-komplett.

Noen ganger kan vi finne algoritmer som alltid genererer optimal løsning, og som nesten alltid bruker kort tid. Til gjengjeld vil de bruke eksponentiell tid i noen få tilfeller. Slike algoritmer kan være egnede av og til.

I det generelle tilfellet er det vanlig å bruke tommelfingerregler — heuristiske regler — for å generere løsninger. Et eksempel på en heuristisk regel er måten man tenker på i åpningsspillet i sjakk. Man ser først om man kan ta noen brikker, eller om noen brikker står for slag (spesielt kongen!), og hvis ikke det er tilfelle, prøver man å "utvikle offiserer", "ha kontroll over sentrum" og "ikke miste tempo". Det er ikke sikkert at et trekk er optimalt, selv om de heuristiske reglene tilsier det. Men det går langt fortere enn å analysere situasjonen til bunns, samtidig som løsningene stort sett blir gode.

En stor ulempe med heuristikk er at vi ikke har noe kontroll over hvor gode resultater vi får, dvs. hvor langt svaret er fra å være optimalt. En ting er at svaret ikke er optimalt, men mange heuristiske algoritmer kan komme med svar som er *vilkårlig* langt unna det optimale. Det er ønskelig å finne algoritmer som garantert gir et svar som ikke er mer enn f.eks. 50% unna det best tenkelige. Dette lar seg gjøre i noen tilfeller. Andre ganger kan man lage algoritmen med en parameter som styrer hvor gode løsninger man er garantert å få (mot at kjøretiden øker tilsvarende). Dette kalles tilnærmingsalgoritmer, og er svært ønskelig. Dessverre er det få problemer man har funnet slike tilnærmingsalgoritmer for, så det er heuristikk, uten noen form for garanti, som dominerer for praktisk bruk.

1.4 Vanlig strategi for å designe kretser

Før man lager en krets må man ha klart for seg hvilke spesifikasjoner man vil den skal oppfylle. Så må kretsens logiske funksjon bestemmes nøyaktig, slik at man vet hvilke byggestener som skal brukes, og hvordan de skal kobles sammen. Dette kalles logisk design, eller syntese. Resultatet av denne fasen er en nettliste.

Så må man finne ut hvordan transistorene/modulene skal plasseres utover og sammenkobles slik at arealforbruket blir så lite som mulig. Dette er utleggsfasen, som er den som skal behandles i denne oppgaven. Hvis denne fasen skal løses av en datamaskin og kretsen er av en viss størrelse finner man fort ut at det er svært vanskelig å løse hele problemet på en gang. Derfor er det vanlig å dele kretsen opp i mindre deler, moduler, som så blir satt sammen i forskjellige faser. Selv om dette gjør problemet mer håndterlig, er delproblemene man støter bort i på denne måten vanskelige nok, og ofte NP-komplette.

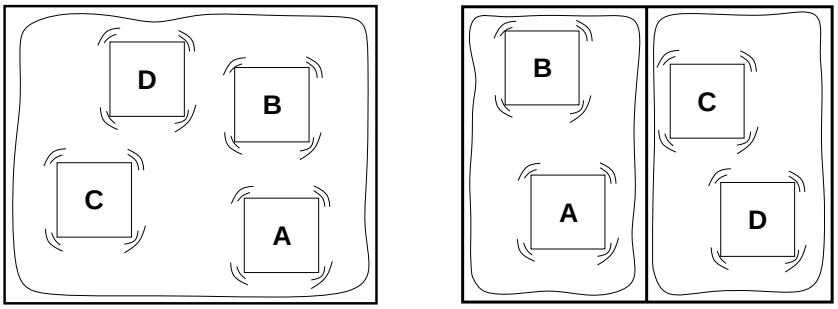
1.4.1 Partisjonering og plassering

Det er ikke vilkårlig hvor på det totale kretsarealet modulene står. To moduler som kommuniserer mye bør plasseres i nærheten av hverandre. Dette vil sørge for at lederne som forbinder

modulene blir kortere, og minske sjansen for at det blir "trafikk-kork" ved at lederne går i veien for hverandre.

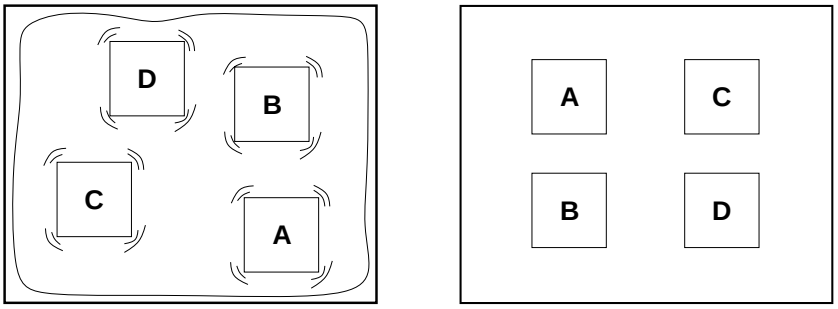
Problemet med å sette modulene på passelige steder kalles plassering (eng. *placement*). Hvis man i tillegg har mulighet til å justere forholdet mellom høyde og bredde (innenfor visse grenser) på modulene, kalles det for "floorplanning". Denne fasen sees ofte på som en to-trinns prosess:

- Partisjonering (eng. *partitioning*) — en grovoppdeling for å finne ut i hvilke moduler som bør havne nær hverandre til slutt. Dette er illustrert i figur 1.6.



Figur 1.6: Partisjonering avgrenser modulenes bevegelsesrom.

- Selve plasseringen — hvor man bestemmer modulenes nøyaktige posisjoner og orientering, eventuelt modulenes forhold mellom høyde/bredde. Plasseringen er illustrert i figur 1.7.



Figur 1.7: Plassering bestemmer modulenes nøyaktige posisjon.

1.4.2 Ruting — sammenkobling av moduler

Modulene må kobles sammen, og den oppgaven kalles for ruting (eng. *routing*). Det å koble sammen mange moduler er vanskelig, så det er vanlig å dele denne oppgaven inn i to faser: globalruting og lokalruting.

I den første fasen, som blir løst av såkalte globalrutere, finner man omtrent hvordan hver forbindelse skal gå. Globalruteren må også finne hvilke områder lokalruteren skal arbeide på.

Lokalruteren får vite nøyaktig hvilket område den skal arbeide på, og hvor lederne går inn og ut fra området. Resultatet er et ferdig utlegg hvor alle detaljer er med.

Området som skal lokalrutes kalles for en koblingsboks (eng. *switchbox*). Hvis det bare er tilkoblingspunkter (terminaler) på to motstående sidekanter, kalles rutingarealet for en kanal (eng. *channel*). Selv om problemet med å rute alle forbindelsene i en kanal er NP-komplett (Szymanski, 1985), er det enklere å rute kanaler (spesielt rektangulære kanaler) enn å rute (generelle) koblingsbokser. Derfor er det et mål for globalruteren å generere så få koblingsbokser som mulig, og heller lage flere kanaler.

1.4.3 Hierarkisk design

Hvis det er en stor krets som skal lages, brukes "splitt og hersk"-teknikken rekursivt, under navnet hierarkisk design. Men et sted må rekursjonen selvfølgelig stoppe, og på det nederste nivået (hvor man skal plassere transistorene) må man bruke en annen metode. Her snakker man noen ganger om transistorene som bladnoder, siden de ikke består av nye moduler.

Disse modulene kan lages manuelt, hentes fra et ferdiglaget modul-bibliotek, eller konstrueres helt eller delvis automatisk av en datamaskin. Et program som konstruerer moduler automatisk kalles for en modulgenerator, og det er det som er temaet for denne oppgaven.

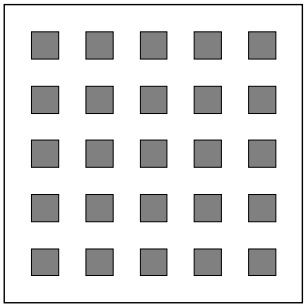
Hvordan transistorene legges ut, er bestemt av hvilken utleggsstil man bruker. Utleggsstilen er delvis bestemt av måten brikkene blir produsert på, og delvis en konvensjon for å gjøre problemet enklere å løse. Her er det flere strategier:

- Programmerbare matriser

  I denne stilen er alle transistorene satt opp i et fast mønster, og oppgaven blir å finne hvilke forbindelser som skal gjøres. Fordelen med dette er at man kan lage mange identiske halvferdige brikker, som blir gitt forskjellige funksjoner etterpå. Man oppnår økonomisk fordel med masseproduksjon, selv om det bare skal lages få av hver type brikke. Det finnes noe varianter av denne stilen, hvor de mest kjente er "gate arrays" og "sea of gates".

- Standardcelle

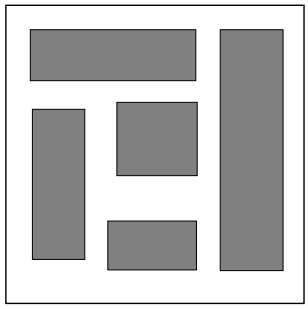
  I denne stilen ligger det små celler regulært utover kretsarealet. Alle transistorene ligger inne i cellene, og all celle-til-celle ruting går utenom cellene. Denne regulære strukturen gjør at utleggsalgoritmene blir enkle, men også at utleggene ikke blir så kompakte som ved bruk av f.eks kundespesifiserte kretser. Se figur 1.8 for et eksempel på denne stilen.



Figur 1.8: Utlegg i standardcelle stil. Cellene, hvor alle transistorene ligger, er grå.

- Makrocelle

  I denne stilen blir det lagt ut moduler, som så blir rutet sammen. Dette er vanlig å bruke på høyere nivåer, selv om man på de laveste nivåene sverger til en av de andre måtene som er beskrevet. Figur 1.9 gir et eksempel på denne stilen.



Figur 1.9: Utlegg i makrocelle stil. Modulene er grå.

- Kundespesifisert (eng. *fullcustom*)

  I denne stilen er ingen ting avgjort på forhånd, og alt tilgjengelig areal blir brukt som det passer. Dette er den dyreste måten å produsere kretser på, men blir gjerne valgt når man trenger noe med spesielle egenskaper eller som skal produseres i store volum. Etterhvert som det kommer verktøy for dette, blir det billigere å bruke denne metoden.

1.4.4 Kompaktering

Når kretsen er ferdig lagt ut, er det ikke sikkert at alt arealet blir utnyttet. Derfor kan man ha en egen kompakteringsfase, hvor man forsøker å gjøre (små) endringer for å få utlegget til å ta mindre plass.

1.4.5 Simulering

Når hele kretsen er laget, bør man simulere den for å forsikre seg at den oppfører seg slik man ønsker. Det kan være flere grunner til at den ikke gjør det. En feil som kan oppstå er at man kan ha gjort direkte feil under arbeidet, enten logiske eller rent praktiske. En annen ting er at man kan ha lagt inn for lange ledere, eller for mange viaer, slik at det blir for store forsinkelser. Dette kan gjøre at kretsen oppfører seg forskjellig fra det man forventet.

Det er vanskelig å finne feil ved simulering og testing, spesielt for manuelt utførte kundespesifiserte utlegg. Generelt er det bedre og billigere å finne feilene så tidlig som mulig, fordi det da blir enklere å rette dem. Dette er et viktig argument for automatisere utleggsprosessen mer, for automatiske verktøy kan ofte hjelpe til med å fange opp feil og unøyaktigheter på et tidlig tidspunkt.

Simulering er en ikke-formell verifikasjonsmetode. Det forskes også på mer formelle verifikasjonsmetoder, hvor man forsøker å bevise at kretsen oppfyller spesifikasjonene. Dette har ikke kommet så langt at det anvendes i noen særlig grad i praksis, men i fremtiden kan det tenkes at dette blir mer vanlig.

Etter at kretsen er simulert og funnet at den oppfyller de kravene man har til hastigheter og oppførsel kan den sendes til prosessering.

1.4.6 Prosessering og testing

Så må man få noen til å prosessere kretsen. Det er flere prosesshus som tilbyr seg å fabrikere kretser utifra ferdig designede utlegg. Det gjelder å velge et prosesshus som er billig, og som tilbyr designregler som er akseptable. Hvis ikke prosesshuset finner noen feil med kretsen, og alt går bra, kan man ha en ferdig brikke etter noen uker.

Når kretsen er ferdig prosessert må den testes, og her er det viktig at man har gode test-data fra designprosessen, slik at man vet hvilke test-data (test-vektorer) som har potensiale for å finne flest feil. For best resultater, bør man designe kretsen med testingen for øye — design for testbarhet.

1.5 Problemstilling og tilnærmingsmåte

1.5.1 Utvikling av et program for å løse utleggsproblemet

Denne oppgaven bygger på ideer av Kjell Øystein Arisland (Arisland, 1989). Han har forsøkt å lage en ny struktur på algoritmene for modulgeneratorer. Først deler man opp nettlisten i mindre deler. For hver del, løser man problemet optimalt, noe som er mulig siden det vil være små instanser på dette nivået. Siden blir delene, minimodulene, satt sammen. Alt dette foregår automatisk, uten interaksjon med brukeren. Arisland skjønte at dette var et ambisiøst prosjekt og ønsket å trekke flere inn i arbeidet.

Min oppgave er å behandle noen sider av prosjektet. En viktig deloppgave er å lage et godt strukturert program for å legge ut de små nettlistene optimalt. Ruting inne i minimodulene blir også behandlet. En annen hoveddel av arbeidet er å gjennomgå den fyldige litteraturen omkring partisjonering/gruppering. Ut fra dette forsøker jeg å formulere et kriterium for partisjonering som gir bedre utgangspunkt for et godt utlegg, enn de kriteriene som er foreslått tidligere. Dette leder til utviklingen av en algoritme som bruker mitt nye kriterium.

1.5.2 Eksterne krav til verktøyet som helhet

Her følger en ekstern spesifikasjon for hele systemet. Som inndata skal programmet ha

- nettliste med angivelse av P- og N-transistorer og eksterne kontakter
- eventuelt ønsket forhold høyde/bredde (med angivelse av et godtagbart intervall)
- eventuelle ønsker om hvor de eksterne kontaktene skal plasseres
- teknologiregler — en høynivå beskrivelse av designreglene som skal brukes

Som utdata skal programmet lage et ferdig rektangulært utlegg, hvor forholdet mellom høyde og bredde er tilnærmet som ønsket, og med eksterne kontakter på angitt plass. I tillegg til å holde seg innenfor kravene som er gitt i designreglene, skal utlegget oppfylle blant annet følgende hovedkriterier:

- bruke så lite areal som mulig
- ha korte lederlengder
- ha få viakontakter

Det viktigste kriteriet er å bruke lite areal, men de andre kriteriene blir også tatt hensyn til i den grad det er praktisk. Delvis blir de også et biprodukt av et lite areal. Det er også andre kriterier, men disse er de viktigste.

Man kan også tenke seg at noe av arealet er opptatt før man begynner plasseringen. Dette kan være ledere som allerede er trukket på tvers av modulen, eller det kan være at modulen blir plassert inntil et område som er ferdig lagt ut. Dette vil imidlertid ikke bli behandlet i denne oppgaven.

Nøyaktig hvor store nettlister man skal kunne klare å legge ut i rimelig tid, kan man ikke si sikkert før man måler ytelsen på det ferdige programmet. Et anslag er at det skal være praktisk å legge ut nettlister med ca 200 transistorer.

1.5.3 En skisse over verktøyets arbeidsmåte

Først deles nettlisten opp i mindre deler (med ca 5 transistorer i hver del). I denne fasen forsøker man å få transistorer som har mye med hverandre å gjøre til å havne i samme del. Nøyaktig antall transistorer som havner i hver del må delvis bestemmes ved praktiske forsøk (hvor mange transistorer kan man ha slik at man kan legge ut en minimodul optimalt i rimelig tid), og delvis hvor mange deler vi ønsker å lage (som igjen er styrt utifra kravet om ønsket forhold mellom høyde/bredde).

Vi må ha en plasseringsfase hvor delene, minimodulene, blir plassert utover i et rutemønster. Nøyaktig hvilken type rutemønster man velger er avhengig av ønsket høyde/bredde-forhold. Det er ønskelig at minimoduler som kommuniserer mye plasseres i nærheten av hverandre, slik at det blir enkelt å rute utlegget, slik at man får korte lederlengder, og slik at man ikke må sette av mer plass for å få rutingen til å gå opp. Rutemønsteret innvirker også på hvor brønner og brønnkontakter bør ligge, samt hvor det er hensiktsmessig å ha ledere for spenningsforsyning og jord.

De små nettlistene blir lagt ut til minimoduler av et delprogram. Nettlistene vil være så små at det skal være mulig å gjøre utlegget optimalt, eller ihvertfall svært nært optimalt. Det er i denne fasen at teknologireglene virkelig spiller inn. Dette delprogrammet foretar plassering og detaljert ruting, og forsøker å lage et så lite utlegg som mulig. Dersom det ikke er mulig å få til et utlegg innenfor slike små grenser, vil grensene bli utvidet helt til det går.

Minimodulene må settes sammen til et sammenhengende utlegg, ved å koble sammen alle forbindelser mellom moduler.

1.5.4 Om resten av oppgaven

Den relevante litteraturen for denne oppgaven blir gjennomgått i kapittel 2. Det er veldig mye litteratur på området, så jeg er nødt til å begrense utvalget noe. Jeg har tatt med mye av det som er gjort innenfor partisjonering, men bare plukket ut det aller mest relevante innenfor ruting.

I kapittel 3 beskriver jeg mitt eget arbeide. Det viktigste er beskrivelsen av Cell, et program for å plassere transistorer, og arbeidet med kriterier og algoritmer for partisjonering.

Kapittel 4 inneholder drøfting av erfaringer og resultater. Dette leder til konklusjonen i kapittel 5. Siste kapittel dreier seg om videre arbeid og muligheter. Til slutt kommer listen over referanser og vedlegget med kildekoden for programmet som plasserer transistorer, samt for programmet som partisjonerer nettlisten opp i mindre deler.

Kapittel 2

Litteraturgjennomgang

I dette kapittelet skal jeg gjennomgå aktuell litteratur for å løse utleggsproblemet.

2.1 Ulike måter å strukturere modulgeneratorer på

Dette avsnittet vil se på forskjellige strategier som er brukt for å strukturere problemet. Jeg begrenser meg til å snakke om det som er mest relevant, nemlig systemer for å legge ut transistorer direkte. Systemer for å legge ut større moduler vil altså ikke bli behandlet her. Systemer som tar utgangspunkt i en funksjonell beskrivelse av kretsen (og ikke i nettlisten), vil heller ikke bli gjennomgått.

Den overordnete strukturen har mye å si for hvilke algoritmer som senere kan brukes. Her må man også gjøre valg som går på om man skal gjøre alt automatisk, eller om man skal la verktøyet være delvis interaktivt.

2.1.1 Arislands arbeid — grunnlaget for denne oppgaven

Denne oppgaven bygger hovedsaklig på arbeidet som er gjort av Kjell Øystein Arisland (Arisland, 1989). Dessverre er ikke artikkelen publisert, så jeg skal beskrive det her.

Først blir problemet analysert for å finne ut hvorfor manuelt utlegg fremdeles ofte blir bedre enn automatisk maskinelt utlegg. Løsningen som blir presentert er å bruke datamaskinenes evne til hurtig å prøve mange muligheter. Det går ikke å utnytte hastigheten direkte, fordi det totale antall muligheter er for stort selv for raske maskiner. Men ved bruke følgende metoder vil man forhåpentligvis få kontroll med problemet:

- Hierarkisk dele opp problemet i så små biter at de lar seg håndtere automatisk ved prøving og feiling.
- Ved å representere utleggsarealet som en matrise eller et rutemønster (eng. *grid*) — slik at det bare er visse punkter som kan brukes til å plassering — fjernes unødvendige detaljer, samtidig som det er rom for å legge inn ganske komplekse teknologispesifikasjoner basert på erfaringer fra manuell design.
- Avanserte avskjæringer slik at håpløse plasseringer ikke blir fulgt opp mer enn nødvendig.

Ved å lese inn et sett med teknologiavhengige regler som beskriver hvordan forskjellige plasserte elementer påvirker hverandre, kan man oppnå å la programmet bli uavhengig av spesifikke teknologier, samtidig som man kan dra nytte av erfaringer fra manuelle utlegg.

Elementene som plasseres kan være transistorer, kontakter og ledere. Transistorer er ikke symmetriske (sett fra nettlistens standpunkt), så man har fire forskjellige elementer som svarer til de forskjellige retningene som en transistor kan stå i. De teknologiavhengige reglene vil bestemme om to elementer er i kontakt eller ikke.

Med dette rammeverket på plass beskrives algoritmene for å styre prosessen, og erfaringer som er gjort i implementasjonen. En viktig erfaring er at arbeidet med å finne et detaljert utlegg er såpass vanskelig at mye av programmet er lagt opp rundt det. For eksempel blir rutinen ikke bedt om å finne det minste rektangelet som lar det blir et utlegg. Isteden får rutinen presentert et utlegg, og skal bare svare hvorvidt det lar seg finne et utlegg innenfor den plassen som er til rådighet. Det blir også sagt noe om hvilket potensiale denne metoden ser ut til å ha, ved å sammenligne med manuelle utlegg.

2.1.2 Talib

Talib (Kim, McDermott, & Siewiorek, 1984) er et verktøy som automatisk plasserer og ruter fra 4 til 86 transistorer i en NMOS prosess med ett metall-lag. Det er et regelbasert ekspertsystem, med over 2000 regler. Noe uvanlig blir ruting og plassering utført samtidig. Talib oppnår stort sett bedre resultater enn ferske designere, og stort sett noe dårligere enn erfarne designere. Kjøretiden er avhengig av hvor mange ganger reglene blir utført. Dette er igjen avhengig av antall transistorer, og tilleggskrav til utlegget som f.eks maksimal bredde og ønsket posisjon av I/O-terminaler.

2.1.3 TOPOLOGIZER

TOPOLOGIZER (Kollaritsch & Weste, 1985) er et ekspertsystem for å legge ut transistorer, som visstnok håndterer opp til ca 20 stykker. Den genererer symbolske utlegg, så for å få detaljerte utlegg som tar hensyn til designreglene blir cellene tatt hånd om av et annet program.

Plasseringen av transistorer ligner på spillet domino, hvor det er om å gjøre å få forskjellige brikker til å passe sammen. Transistorer har tre tilkoblingspunkter (ikke to som dominobrikker), så det er vanskelig å lage en helt tett plassering, men det er heller ikke nødvendig for å rute utlegget. Programmet starter med en tilfeldig plassering, og forbedrer det ved å bruke regler som bytter om på transistorene slik at de skal passe bedre med nabotransistorene.

Rutingen blir gjort i to faser. Først kalles det på en konstruktiv algoritme (forskjellen på iterative og konstruktive algoritmer blir forklart på side 23) som finner en løsning som er korrekt, men ikke nødvendigvis kompakt. Etterpå blir løsningen iterativt forbedret, f.eks ved å endre en leder som har en U-form til å gå rett over hvis det er mulig.

2.1.4 BBC

BBC (Hughes, Salama, & Liu, 1986) er en modulgenerator for en CMOS prosess med to metall-lag, og bruker en utleggstil med celler plassert i rader. Ved å sørge for at naboceller har samme type transistorer mot hverandre (dvs. N til N og P til P) kan kraft og jord alternere mellom cellene.

Inne i cellene blir transistorene plassert så tett som mulig, ved at source fra en transistor settes direkte inntil drain på en annen. Dette kalles *abutting* på engelsk, og kan selvsagt bare gjøres dersom de tilhører samme nett. Hvis nettet bare består av de to transistorene kan de

settes svært tett, men det er mye å hente selv det må plasseres en kontakt mellom transistorene for å la nettet bli koblet til andre steder. I artikkelen blir følgende tabell presenterert som motivasjon:

Separation	Condition
14 λ	Non-abutting source and drain
2 or 3 λ	abutting source and drain with no contact
4 or 6 λ	abutting source and drain with contact

BBC bruker Kernighan-Lin algoritmen (se avsnitt 2.3.3) for grafpartisjonering gjentatte ganger for å tilordne transistorene til rader. Ved å legge høy kostnad på P- og N-transistor forbindelser, øker de sannsynligheten for at de kommer i samme rad. Tilsvarende får store nett tilordnet lav kostnad, slik at de lettere skal kunne gå mellom rader.

Når de skal rute mellom poly-kontakter prøver de å bruke et lag (fordi utlegget derfor blir mer kompakt). Derfor bruker de topologisk ruting (nærmere beskrevet i 2.6.5). De har en heuristisk algoritme som kan backtracke (se avsnitt 2.6.2) hvis man oppdager at man er i ferd med å ødelegge for andre forbindelser. Ellers brukes "left edge" algoritmen, som er en metode som brukes i kanalrutere for å minimere arealbruken. Rutingen vil aldri stoppe opp, fordi man i så fall ekspanderer størrelsen på rutingarealet.

2.1.5 EXCELLERATOR

EXCELLERATOR (Poirer, 1987) er et verktøy for plassering og detaljert utlegg av transistorer. I eksemplene er det håndtert fra 12–28 stykker. Tidlige versjoner av EXCELLERATOR brukte en regelbasert fremgangsmåte a la Topologizer, men det viste seg å bli for komplisert å håndtere alle spesialtilfeller på den måten.

Derfor blir transistorene plassert av en grådig søkealgoritme, med lite bruk av regler. Et av målene for denne søkealgoritmen er at transistorene skal settes direkte inntil hverandre (tilsvarende slik det er gjort i BBC). Programmet tilbyr forøvrig en viss støtte for store transistorer ved å kopiere opp et multippel av transistorer med minimum størrelse.

Rutingfasen bruker en avansert rekursiv algoritme, som har mulighet til å fjerne tidligere plasserte nett, hvis rutingen stopper opp. Det er tilsvarende Mighty, som blir beskrevet i avsnitt 2.6.6. Når et allerede rutet nett har blitt fjernet fordi det blokkerer et annet, får man ofte problemer med å rute det på nytt. Man risikerer at de forskjellige nettene fjerner hverandre syklisk. EXCELLERATORs løsning på dette er å ta kostnaden med å rerute plasserte nett med i beregningen når et nett har problemer med å rutes. Ved å sørge for å bare fjerne nett som lar seg rute om igjen, unngår man dette problemet.

2.1.6 CLAY

I (Kollaritsch, Lusky, Prasad, & Potter, 1988) beskrives CLAY — en pakke for å gå fra nettliste og en teknologifil, til ferdig utlegg. Systemet er ment å klare opp til 2000 transistorer. I eksemplet som er gitt blir 54 transistorer delt opp i 6 deler, hvor den største inneholder 20 transistorer.

Plasseringsdelen ligner på den i Topologizer. Hele modulen er legges ut på et stort rutemønster, og ruting blir gjort på hele modulen under ett.

2.1.7 CLEO

CLEO (Domic, Levitin, Phillips, Thai, Shiple, Bhavsar, & Bissell, 1989) er et verktøy som legger ut 25–500 transistorer. Det legger vekt på å være fleksibel med hensyn på begrensninger på høyde/bredde-forhold og tidligere plasserte elementer. Mye kan styres av brukeren, og CLEO vil heller lage et uferdig utlegg enn å bryte med kravene fra brukeren. Programmet legger ut "logiske porter" (NAND, OR, osv.), og ikke transistorer. Som i BBC blir rekursiv min-kutt brukt for plassering. Til slutt blir det brukt et bibliotek for å konvertere logiske porter til transistorer, men programmet kan også konvertere automatisk vha simulert størkning (se side 28). Det tar ca 1 minutt CPU tid for flere hundre porter.

2.1.8 CETUS

(Sun, 1989) beskriver CETUS, som er et system for å legge ut transistorer. I eksemplene er det lagt ut fra 8 til 26 stykker. Vi ønsker at diffusjonen fra en transistor skal ligge tett inntil diffusjonen fra en annen. For få dette til i stor skala, må vi

CETUS prøver å maksimalisere antall ganger man kan sette transistorene helt tett inntil hverandre (slik som beskrevet for BBC og EXCELLERATOR), ved å bruke "Eulergraf" metoden.

For å rute utlegget blir det brukt en variant av Mighty (se avsnitt 2.6.6) modifisert til å klare tre lag.

2.1.9 CELLO

Cello (Madsen, 1989) er et system for å legge ut transistorer i CMOS teknologi. I eksemplene blir det lagt ut 12–22 stykker. I likhet med CETUS, blir "Eulergraf" metoden brukt til å finne riktig ordning på transistorene.

Hvis man antar at man ikke trenger å holde seg strengt til nettlisten, sålenge den endelige kretsen utfører den samme funksjonen, kan man få til noen optimaliseringer. Siden rekkefølgen i nettlisten innvirker på ytelsen til kretsen, er dette noe man må avklare før man begynner å endre på den. Cello bruker algoritmer som kan endre på nettlisten, eller la være, avhengig av hva man spesifiserer.

2.1.10 Hoyle et als ekspertsystem for plassering

I (Hoyle, Priest, & Hetherington, 1991) blir det presentert et ekspertsystem for å plassere bladnoder (dvs. 10–100 transistorer). Systemet foretar kun plassering, og ikke ruting. Basert på samtaler med designeksperter hos Texas instruments, har de definert ni regler for å flytte på transistorer slik at utlegget skal kunne bli kompakt. En konstruktiv algoritme lager en initiell plassering utifra de samme prinsippene som disse reglene. Deretter blir utlegget forbedret iterativt ved at reglene blir utført. De har forsøkt forskjellige måter å kontrollere regelutførelsen, blant annet simulert størkning (se avsnitt 2.3.3 for mer om den metoden) o.l. Dette fører til bedre løsninger, men dårligere ytelse, enn ved å bare bruke de konstruktive reglene.

2.2 Oppdeling i faser

De fleste forskere er enige i at man må dele opp i faser. Men det er ikke helt fastlagt hvilke faser det er lønnsomt å ha. Her følger en liste over faser som er vanlige for å løse utleggspro-

blemet:

- Partisjonering
- Plassering/Floorplanning
- Modulplassering
- Globalruting
- Lokalruting
- Elementplassering

Egentlig er disse fasene ikke ønskelige, fordi de fjerner endel lovlige løsninger fra løsningsrommet, deriblant muligens den optimale. Grunnen til at vi allikevel deler opp i faser, er at vi ikke klarer å håndtere hele problemet på en gang. Men selv om vi har innfunnet oss med å løse problemet i faser, er det viktig å prøve å analysere hvilke faser vi skal ha, og hvor skillet mellom fasene skal gå. Det er en trend i retning av at de forskjellige fasene glir over i hverandre, eller ihvertfall at det blir mer kommunikasjon mellom forskjellige faser. Et eksempel på det er arbeidet her ved instituttet med å knytte globalruting og lokalruting tettere sammen (Knudsen, 1991; Næss, 1991). Et annet eksempel er integrasjon mellom floorplanning og globalruting (Lengauer avsnitt 8.8). Ved å kombinere faser, kan man ta hensyn til faktorer som kan bety mye for andre faser i situasjoner hvor man ellers ville ha gjort et vilkårlige valg fordi det ser uviktig ut i den aktuelle fasen.

Men hvis man knytter to eller flere faser for tett sammen, har man gått sirkelen helt rundt, og er tilbake ved grunnen til at man innførte faser i først omgang: Man klarer ikke å lage algoritmer som løser så komplekse problemer innen rimelig tid. Derfor er viktig å finne et grensesnitt mellom fasene som gjør at man får så god kommunikasjon som mulig, uten at fasene integreres for sterkt.

2.3 Partisjonering

Dette avsnittet inneholder en oversikt over det som er gjort med hensyn på dele opp en graf i to eller flere deler. Først skal jeg se på noen problemdefinisjoner for bipartisjonering av grafer, og så algoritmer som er foreslått for å løse de problemene.

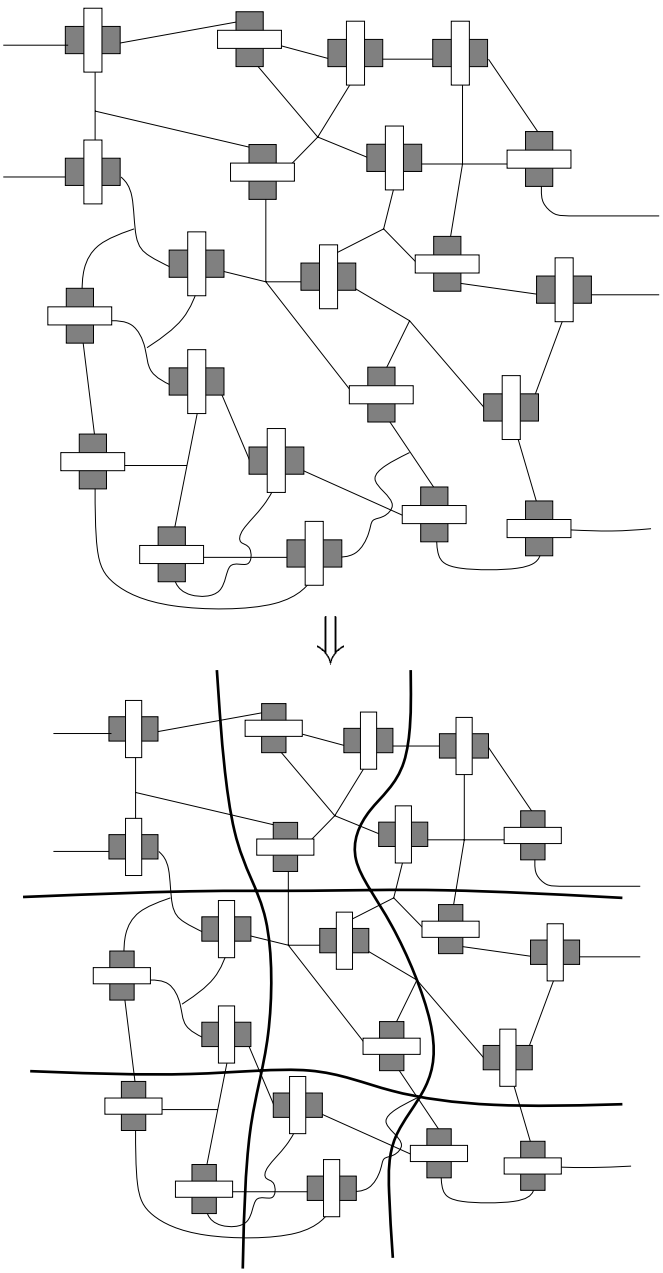
Jeg ser også på utvidelser til hypergrafer. I hypergrafer kan en kant kan gå mellom mer enn to noder, slik som et nett kan koble sammen flere transistorer.

Tilslutt ser jeg på oppdeling i mer enn to deler. Dette kan kalles multipartisjonering, k-partisjonering eller gruppering.

2.3.1 Problemdefinisjon

Problemet er å foreta en grovplassering. Med grovplassering mener jeg at man ikke trenger å finne en helt eksakt plassering, man skal bare finne et område som de aktuelle elementene skal plasseres innenfor.

Som inn-data skal denne fasen få en nettliste med opptil 200 transistorer, og som resultat ca 40 nettlister med ca 5 transistorer. Figur 2.1 er en forenkling fordi en virkelig nettliste normalt ikke lar seg legge ut i et plan så godt som her, men det skulle vise hva vi er ute etter.



Figur 2.1: Partisjonering av nettliste.

En annen grunn til at dette er en forenkling, er at siden vi opererer med CMOS teknologi, må alle transistorene av en type (P eller N) ligge i brønner, mens resten må ligge utenfor (eller i en annen type brønn). Som nevnt i avsnitt 1.2.4, er det vanlig å samle flere transistorer i en brønn. Ellers vil mye areal går med til sikkerhetssoner og brønnkontakter. Derfor er det fristende å legge på et ekstra krav til grovplasseringen, om at det bare skal være en transistortype i hver del.

For å komme til kjernen av problemet, skal jeg først se på en enklere problemstilling: Oppgaven med å dele grafer (og ikke nettlister) i to deler er vanskelig nok. Hvis dette skal løses optimalt, må man først finne det optimale utlegget hvor vi tar hensyn til alt som er

relevant (f.eks. nøyaktig hvilke designregler som skal brukes). Så kan utlegget deles opp, og informasjonen som sier hvilke transistorer som ligger i hvilken del returneres. Det sier seg selv at dette ikke er praktisk gjennomførbart. Det er jo nettopp fordi vi ikke kan lage et program som genererer det optimale utlegget at vi ønsker å bruke partisjonering som en del av løsningsmetoden.

Det er naturlig å sette seg litt lavere mål for hva vi ønsker at partisjoneringen skal ta hensyn til. En ting som er klart, er at transistorene som har mye med hverandre å gjøre sannsynligvis bør ligge nær hverandre. Hvor mye de transistorene har med hverandre å gjøre, er bestemt utifra nettlisten. Så det er rimelig å basere partisjoneringen hovedsaklig på informasjonen man finner der. Jeg ser altså bort fra detaljene omkring designreglene i denne fasen.

Jeg skal se på noen problemformuleringer, og se hvor godt de stemmer overens med det problemet som faktisk skal løses.

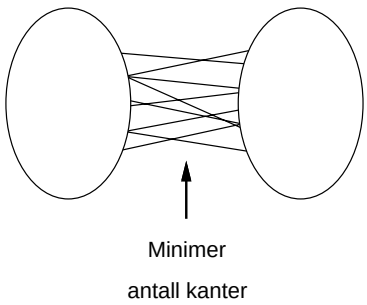
2.3.2 Problemformulering for bipartisjonering

Det enklere problemet med å dele en graf i to deler kalles bipartisjonering. Jeg skal se på noen forskjellige kriterier for bipartisjonering, og starter med den vanligste (som er så vanlig at noen bruker bipartisjonering synonymt med minimum kutt-kriteriet).

Minimum kutt-kriteriet

Hvis vi på forhånd bestemmer oss for hvor mange noder vi ønsker i hver del (f.eks halvparten av nodene), har vi et problem vi kan kalle minimum kuttproblemet, eller bare min-kutt. (Her er terminologien litt flytende, noen bruker "min-kutt" om problemet, andre bruker det om algoritmen til Kernighan og Lin for å løse problemet. Jeg refererer til kriteriet som "min-kutt", og algoritmen som "KL".)

Vi ønsker å minimere antall kanter som går mellom — "blir kuttet av" — de to delene av grafen, som illustrert i figur 2.2.



Figur 2.2: Bipartisjonering: Minimaliser antall kanter som går mellom de to delene av grafen. Figuren er basert på en tilsvarende fra (Dunlop & Kernighan, 1985).

For å snakke om NP-kompletthet må vi formalisere problemet mer, og samtidig må vi formulere det som et desisjonsproblem. En mulig formulering er følgende:

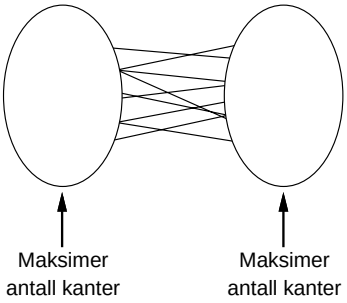
Instans: Graf $G = (V, E)$ hvor V er mengden av nodene og E er mengden av kantene, positive heltall K og J.

Spørsmål: Finnes det en oppdeling av V i disjunkte mengder V_1 og V_2 slik at $|V_1| = K$ og slik at hvis $E' \subseteq E$ er mengden av kanter som har endepunktene sine i hver sin mengde så er $|E'| <= J$.

Selv dette forenklede problemet er NP-komplett. Artikkelen hvor dette ble gjort, (Hyafil & Rivest, 1973), er vanskelig å få tak i, så jeg har ikke selv lest den.

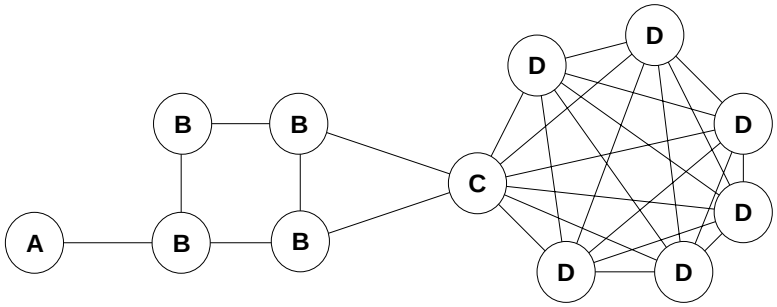
En annen måte å formulere problemet på som konsentrerer oppmerksomheten om innsiden av delene, er å si at istedenfor å minimalisere antall kanter som blir kuttet, ønsker vi å maksimalisere antall kanter som går mellom noder i samme del. Dette kriteriet er illustrert i figur 2.3. Dette gir det samme resultatet, fordi det totale antall kanter er summen av interne kanter og eksterne kanter. Det å maksimere antall interne kanter, er derfor ekvivalent med å minimere antall eksterne — siden det totale antall kanter er uforandret.

Grunnen til at jeg allikevel nevner dette, er at det lar oss konsentrere oss om andre sider ved problemet. Hvilken synsvinkel vi velger får dessuten mye mer å si når vi går fra å dele i to, til å dele i flere deler.



Figur 2.3: Bipartisjonering: Maksimaliser antall interne kanter i de to delene av grafen.

Optimal oppdeling av grafen i figur 2.4 med min-kutt som kriterium gir seks kuttede kanter ved å plassere nodene merket D i en del, og alle de andre i den andre delen.



Figur 2.4: Min-kutt (med like mange noder i hver del) deler opp grafen med kostnad 6.

Minimum kutt-kriteriet, med tillatt avvik

Det finnes også en variant av min-kutt-problemstillingen, som letter litt på kravet om at antall noder på hver side skal være nøyaktig bestemt på forhånd (Fiduccia & Mattheyses, 1982). De innfører en ny parameter r, som sier hvor langt unna de ønskede antall noder i hver del vi

kan være, slik at vi fremdeles godtar løsningen. Hvis to løsninger har samme kuttverdi, blir de betraktet som ekvivalente, uavhengig av antall noder i hver del (sålenge ikke avviket er større enn r).

Hvis tillatt avvik, r, settes til 0, blir dette ekvivalent med vanlig min-kutt, og derfor er denne varianten også NP-komplett.

Hvordan grafen i figur 2.4 vil bli delt med dette kriteriet er helt avhengig av verdien til r.

Maksimal flyt, minimum kutt

Istedenfor å bestemme seg for antall noder på hver side av fordelingen på forhånd, kan vi be om å finne den oppdelingen som kutter færrest kanter totalt sett. Formelt:

Instans: Graf $G = (V, E)$, positivt heltall J.

Spørsmål: Finnes det en oppdeling av V i disjunkte mengder V_1 og V_2 slik at hvis $E' \subseteq E$ er mengden av kanter som har endepunktene sine i hver sin mengde så er $|E'| <= J$.

Dette kriteriet kalles maksimal flyt, minimum kutt (eng. *max-flow min-cut*), forkortet til maks-flyt-min-kutt. Det fine med dette kriteriet, er at det kan løses i polynomisk tid. Algoritmene for å gjøre det blir beskrevet på side 33.

Et fundamentalt problem med dette kriteriet er at man risikerer å få veldig skjeve fordelinger. Det er jo ikke særlig nyttig å vite at man kan dele med bare et kutt, ved å la en enslig node ligge i en del mens alle de andre ligger i den andre. Den minste kuttverdien vil alltid være mindre eller lik antall kanter på den noden som har færrest kanter.

På grafen i figur 2.4 vil maks-flyt-min-kutt-kriteriet dele slik at det bare er node A alene i den ene delen. Dette vil bare kutte et nett, men vil ikke være spesielt nyttig.

Forholdskutt

Et kriterium som er en slags mellomting mellom bipartisjonering og maks-flyt-min-kutt, er forholdskutt (eng. *ratio cut*). Det sier at det ikke er ønskelig at fordelingen blir veldig skjev, men det er ikke kritisk at man får en helt eksakt oppdeling i to heller. Måten å balansere disse to kravene er å si at vi ønsker å minimalisere

$$\frac{|E'|}{|U| \times |W|}$$

Telleren favoriserer lav kuttverdi (som i maks-flyt-min-kutt), mens nevneren favoriserer jevn størrelse på delene, ved at $|U| \times |W|$ er størst når $|U| = |W|$.

Formelt blir dette:

Instans: Graf $G = (V, E)$, positivt heltall J.

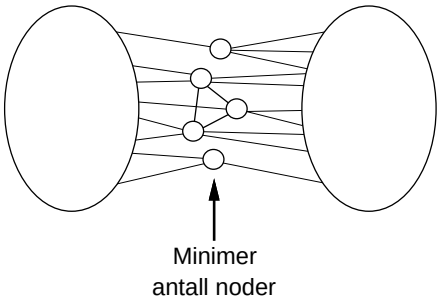
Spørsmål: Finnes det en oppdeling av V i disjunkte mengder V_1 og V_2 slik at hvis $E' \subseteq E$ er mengden av kanter som har endepunktene sine i hver sin mengde så er $\frac{|E'|}{|U| \times |W|} <= J$.

Dette ble foreslått uavhengig av (Wei & Cheng, 1989) og (Leighton & Rao, 1988). Dette kriteriet er også NP-komplett, som vist i (Wei & Cheng, 1991).

Eksempelgrafen i figur 2.4 vil bli delt med A og B nodene (fem stykker) i den ene delen og C og D nodene (syv stykker) i den andre ved bruk av forholdskutt-kriteriet. Dette kutter to nett, og gir forholdskuttverdi på $2/5 \times 7 \approx 0.057$.

Partisjonering ved å fjerne noder

En annen måte å splitte opp en graf på, er ved å plukke ut et antall noder, slik at de gjenværende nodene havner i to deler som ikke har noen felles kanter. Mengden av nodene som plukkes ut kalles en separator, og problemet består i å finne en oppdeling som har så få noder som mulig med i separatoren. Det er også vanlig å kreve at det antall noder i de to delene ikke overstiger en angitt grense.



Figur 2.5: Minimaliser antall noder som må fjernes for at resten av grafen blir delt i to deler som ikke har noen felles kanter.

Dette problemet er NP-komplett problem som beskrevet i (Bui, Chaudhuri, Leighton, & Sipser, 1987).

Den optimale separatoren for eksempelgrafen i figur 2.4 består av noden C. Hvis man ikke har lagt inn så sterke krav på størrelsen på de to delene, kan også muligens B-noden nærmest A være en løsning.

Dette problemet har ikke blitt brukt direkte til VLSI, men algoritmer for å løse problemet kan brukes som grunnlag for å løse f.eks min-kutt.

2.3.3 Algoritmer for bipartisjonering

Når jeg i det følgende beskriver algoritmer, vil jeg klassifisere dem etter følgende mønster:

- Konstruktiv. Bygg opp en løsning fra grunnen.
- Iterativ. Ta en løsning og forbedre den. Kan starte med en tilfeldig løsning, eller en som er bygget opp konstruktivt.

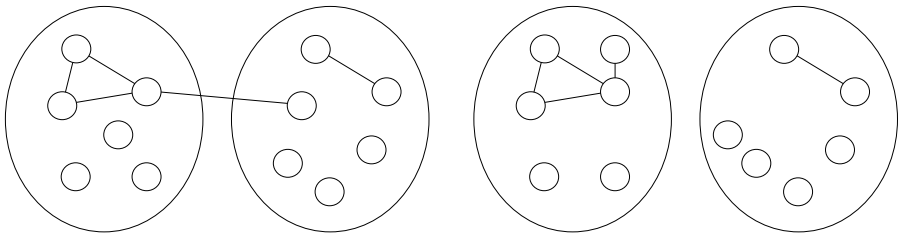
Dette sier litt om fremgangsmåten i algoritmen, og vil også kunne gi et hint om at man muligens kan kombinere to metoder ved å først kjøre en konstruktiv algoritme, og så fortsette med en iterativ.

En enkel konstruktiv algoritme

Man kan rett og slett foreta delingen tilfeldig. Dette gir selvsagt svært lite optimale resultater. Men hva om vi foretar la oss si 100 slike tilfeldige delinger, og velger den som er best? Det viser seg at også dette gir svært dårlige resultater, fordi det er veldig mange måter å dele på, og bare et fåtall av disse er i nærheten av det optimale.

En enkel iterativ algoritme

I stedet for å stoppe med en tilfeldig oppdeling, kan vi se om vi kan forbedre den litt. Vi kaller den ene partisjonen for A, og den andre for B. For alle par av noder, $(a, b), a \in A, b \in B$, finn reduksjonen i antall nett som blir kuttet hvis a og b bytter plass. Velg det paret som gir størst reduksjon i antall kuttede nett, og foreta en ombytting. Gjenta dette til det ikke lenger finnes noen par som gir reduksjon hvis de ombyttes.

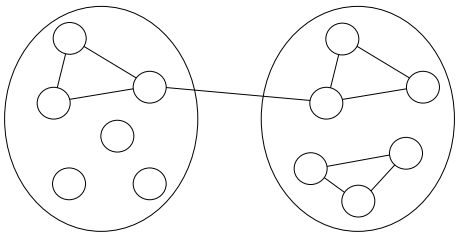


(a) Før byttet, kuttverdi 1.

(b) Etter byttet, kuttverdi 0.

Figur 2.6: Den enkle iterative algoritmen i bruk.

Dette er en grei metode, men prosessen har alt for lett for å stoppe opp. For eksempel vil den ikke kunne forbedre den følgende oppdelingen:



Figur 2.7: En starttilstand som ikke forbedres med den enkle iterative algoritmen.

"Kernighan og Lin"-algoritmen (KL)

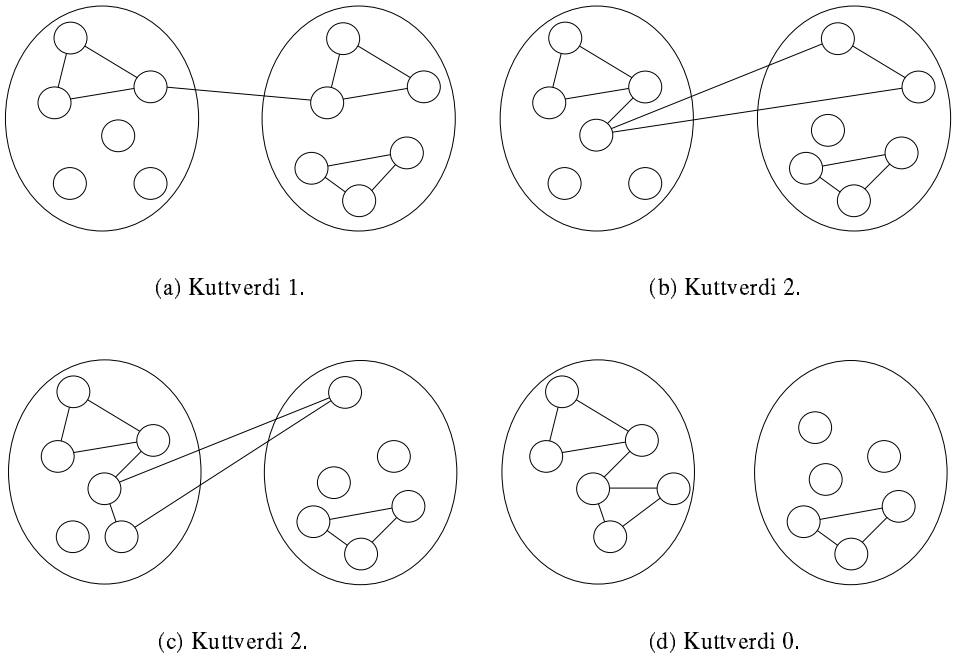
I 1970 kom (Kernighan & Lin, 1970), som introduserte en algoritme som kan sees på som en forbedring av den iterative jeg skisserte over. Algoritmen kalles "Kernighan og Lin"-algoritmen, forkortet til KL, og har dannet grunnlaget for mange senere algoritmer. (Noen kaller den også min-kutt-algoritmen, men det bruker jeg som navn på problemet algoritmen løser.) Isteden for å stoppe med en gang man ikke lenger får en reduksjon i antall kuttede nett, er det nå mulig at man midlertidig ser på ombyttinger som gir et dårligere resultat. Dette organiseres ved at ombyttinger skjer i runder. I hver runde gjør vi følgende:

- Finn paret som gir sterkest reduksjon (eller minst forverring) av antallet kuttede kanter.
- Bytt om disse.
- Sørg for at nodene som byttet plass ikke blir med på flere bytter denne runden.

- Gjenta denne prosessen til det ikke lenger er noen noder som ikke har vært innblandet i noen bytter.
- Velg den konfigurasjonen som har kuttet færrest kanter til nå.

Sett igang en ny runde dersom forrige runde førte til en forbedring.

Denne algoritmen stopper ikke opp så lett som den iterative algoritmen, og vil f.eks. forbedre grafen i figur 2.7, som demonstrert i figur 2.8.

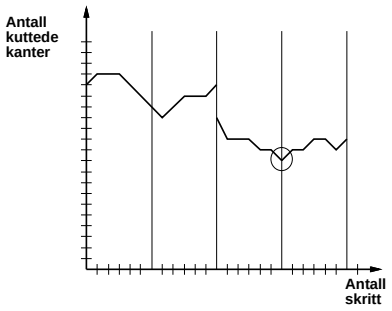


(a) Kuttverdi 1. (b) Kuttverdi 2.

(c) Kuttverdi 2. (d) Kuttverdi 0.

Figur 2.8: KL finner den optimale løsningen i løpet av en runde. Figuren er basert på en tilsvarende fra (Dunlop & Kernighan, 1985). (Deres figur viste feilaktig en graf som KL ikke ville ha forbedret, nemlig den man for ved å fjerne de tre kantene som er litt tynnere enn de andre.)

Her så vi at algoritmen finner minimum i løpet av en runde, slik at det ikke er noe å tjene på å starte en ny runde. Generelt, med litt større grafer, vil man kunne få forbedringer i noen runder — forbedringene stopper gjerne etter omtrent fem runder. Dette er forsøkt illustrert i figur 2.9.



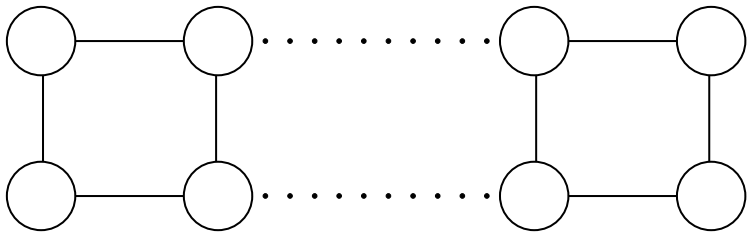
Figur 2.9: En graf (plott) med typisk forløp av KL. Når vi starter en ny runde (markert med vertikale streker) vil vi alltid velge den oppdelingen vi har sett som gir lavest antall kuttede kanter, slik at grafen kan bli diskontinuerlig (som ved starten av runde tre). Den beste oppdelingen er markert med en sirkel.

Problemer med KL Kompleksiteten til KL er litt avhengig av nøyaktig hvordan den blir implementert, men med en normalt god implementasjon vil den ha en orden på $O(n^2 \log n)$. For mange formål er dette litt mer enn ønskelig.

Vanligvis er det mange tilfeller i algoritmen at det blir foretatt valg relativt vilkårlig, hvor den ene alternativet i virkeligheten er mye bedre enn det andre. En slik ikke-determinisme er uheldig, fordi det gjennomsnittlig vil bli foretatt noen dårlige valg i løpet av en kjøring. Hvis algoritmen kunne ha avgjort hvilket alternativ som var best, ville mange dårlige valg vært unngått.

Problemformuleringen som KL løser, krever at antall noder i oppdelingen må bestemmes på forhånd. Siden KL alltid foretar bytter av noder mellom partisjonene, er det vanskelig å tilpasse algoritmen til andre problemformuleringer som løsner litt på det kravet. En måte å gjøre det på, er å legge til noen nye "falske" noder som ikke er tilknyttet noen andre. Når disse nodene blir fjernet til slutt, kan fordelingen blant de gjenværende nodene bli ubalansert.

Det er visse graftyper KL jevnt over gjør det dårlig for. I figur 2.10 ser vi et eksempel på en graftype som i de mange startposisjoner ikke blir forbedret vesentlig.



Figur 2.10: En stigegraf som KL gjør det dårlig på.

Fiduccia og Mattheyses' algoritme

Det har kommet noen artikler hvor KL forbedres. Den første er (Fiduccia & Mattheyses, 1982) som beskriver en variant av KL som sikrer at algoritmen får orden $O(p)$ (p = totalt antall terminaler). Istedenfor å foreta bytter, flytter de enkeltnoder fra den ene siden til den andre, men i tillegg har de et balansekrav for å unngå at alle nodene havner på samme side. Merk at balansekravet gjør at dette ikke lenger er "ren" min-kutt (hvor antall noder i hver del blir bestemt på forhånd), men isteden den varianten hvor man tillater litt avvik. Med disse endringene kan det brukes en datastruktur som gjør at operasjonen som finner neste node som skal byttes går i konstant tid.

Krishnamurthys algoritme

I den foregående algoritmen (i likhet med KL), skjer det ofte at flere noder kan være like gode å flytte over, basert på hvor stor forbedring man oppnår direkte. Men det ene valget vil kunne føre til større fremtidige forbedringer, enn det andre.

Algoritmen beskrevet i (Krishnamurthy, 1984) ser lenger fremover før man velger hvilken node som skal bytte side, og oppnår dermed høyere kvalitet på løsningene.

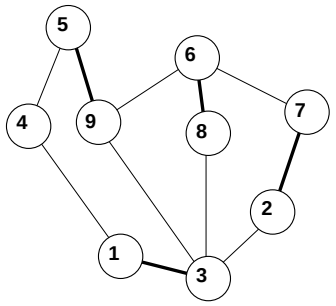
Kjøre KL på en sammentrukket graf

En annen form for forbedring er foreslått i (Bui, Heigham, Jones, & Leighton, 1989). Ved å bruke sammentrekning (eng. *compaction*) forbedres kvaliteten (og av og til ytelsen) på grafer med gjennomsnittlig få kanter per node.

Som et ledd i algoritmen, finner man først en maksimal matching for grafen (jeg vet ikke om noen passende norske oversettelser for matching, så jeg bruker det engelske ordet videre). En matching for en graf er et utplukk av kantene slik at ingen noder i grafen er endepunkt til mer enn en av de utplukkede kantene. En matching er maksimal hvis den ikke kan utvides med flere kanter (slik at det fremdeles er en matching).

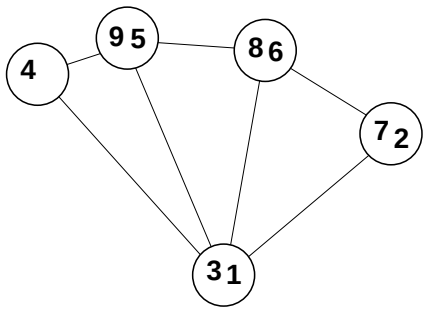
Sammentrekningen foregår slik:

1. Finn en maksimal matching M av G.



Figur 2.11: De markerte kantene er en maksimal matching for grafen.

2. Lag en ny graf G' ved å trekke sammen kantene i matchingen.



Figur 2.12: Slik blir den nye grafen som er basert på matchingen over. Dette eksemplet er litt lite til å vise at den nye grafen normalt har høyere kanttetthet enn den gamle.

3. Kjør KL.

4. Ekspander G' til G, men behold oppdelingen man fikk i 3.

5. Kjør KL med G som starttilstand.

Dette fungerer fordi KL gjør det bedre på grafer hvor kanttettheten er høy. Faktisk vil de aller fleste algoritmer kunne oppnå forbedringer ved en slik metode. Det kan tyde på at en slik sammentrukket graf har færre lokale optima, som ikke også er nær globale optima.

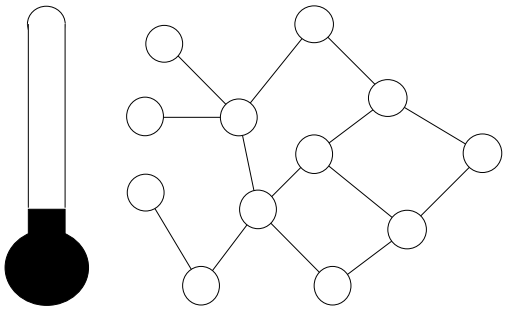
Simulert størkning

En teknikk som har blitt veldig populær for å løse mange optimaliseringsproblemer av den typen man støter på i VLSI-sammenheng kalles simulert størkning (eng. *simmulated annealing*). Jeg bruker den engelske forkortelsen, SA, i fortsettelsen.

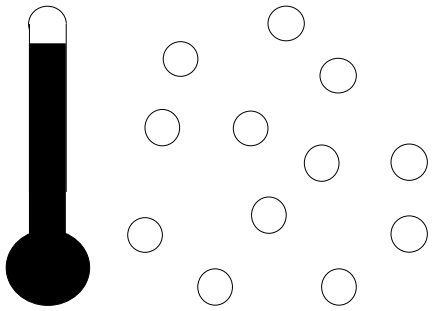
Som navnet antyder, er metoden basert på en analogi til en størkningsprosess, nemlig det som skjer ved størkning av metall.

En ekte størkningsprosess Hvis man har et metallstykke som har ujevn krystallstruktur (f.eks. fordi det har vært utsatt for et sterkt trykk) kan man få tilbake en jevn struktur ved følgende metode:

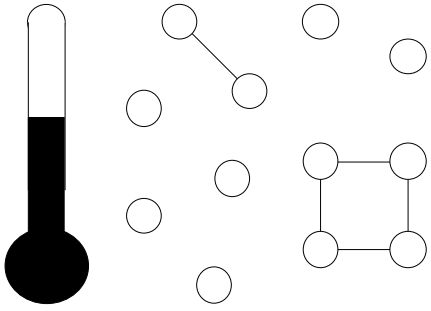
Først oppvarmes metallet til en så høy temperatur at molekylene får nok kinetisk energi til å rive seg løs fra den fastbundne strukturen metallet vanligvis har. Hvis nå materialet avkjøles langsomt, vil atomene gradvis danne en jevn krystallstruktur med lav indre energi. Hvis avkjølingen foretaes for fort risikerer man at atomene ikke får tid til å finne korrekte posisjoner, mens hvis den går for sakte tar det lenger tid enn nødvendig.



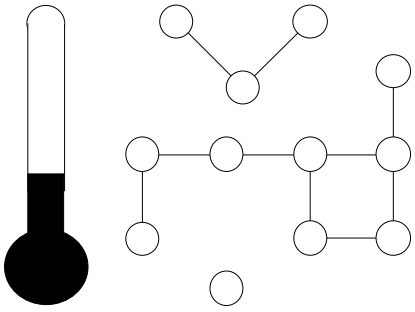
(a) Starttilstand med ujevnt gittermønster.



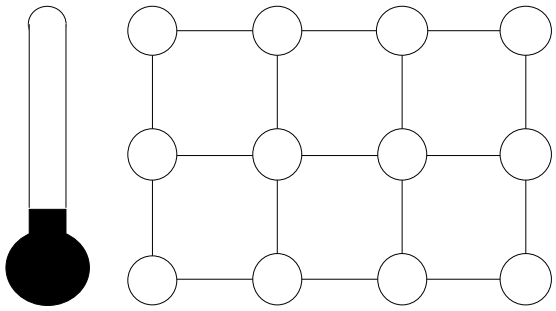
(b) Oppvarming til høy temperatur: bindingene mellom molekylene løses opp.



(c) Temperaturen senkes sakte: noen bindinger oppstår.



(d) Temperaturen synker mer: en jevn struktur begynner å ta form.



(e) Lav temperatur: prosessen er ferdig, og materialet har jevn gitterstruktur

Figur 2.13: De forskjellige tidsfasene i en størkningsprosess.

En simulert størkningsprosess Hva har en slik størkningsprosess med kombinatoriske problemer som partisjonering å gjøre? For metallstykket ønsker vi å finne en atomstruktur som har lav energi, mens for partisjoneringsproblemet vil vi ha en løsning av høy kvalitet.

Metoden baserer seg på at vi foretar tilfeldige ombyttinger. Under simuleringen holder vi greie på "temperaturen". Hvis temperaturen er høy, vil vi godta bytter som gjør kvaliteten dårligere, mens hvis den er lav vil vi sannsynligvis ikke godta det.

Vi starter med høy temperatur, og senker den gradvis. Nøyaktig hvor fort dette skjer er viktig for tiden det tar, og kvaliteten vi får, akkurat som ved en ekte størkningsprosess. Dette er det forsket mye på, og det kalles implementasjonens avkjølingsskjema (eng. *cooling schema*).

En annen måte å se SA på, er som en metode for å komme seg ut av lokale minima. Det er en avansert metode for å styre lokal optimalisering, eller iterativ forbedring. Når temperaturen er høy, er det lett å komme seg ut av lokale minima, men ettersom temperaturen synker, vil det bli vanskeligere å gå til en løsning med dårligere kvalitet. Derfor er det så viktig at avkjølingsskjemaet er tilpasset problemet, slik at man finner et minimumspunkt med god kvalitet, før man slår seg til ro.

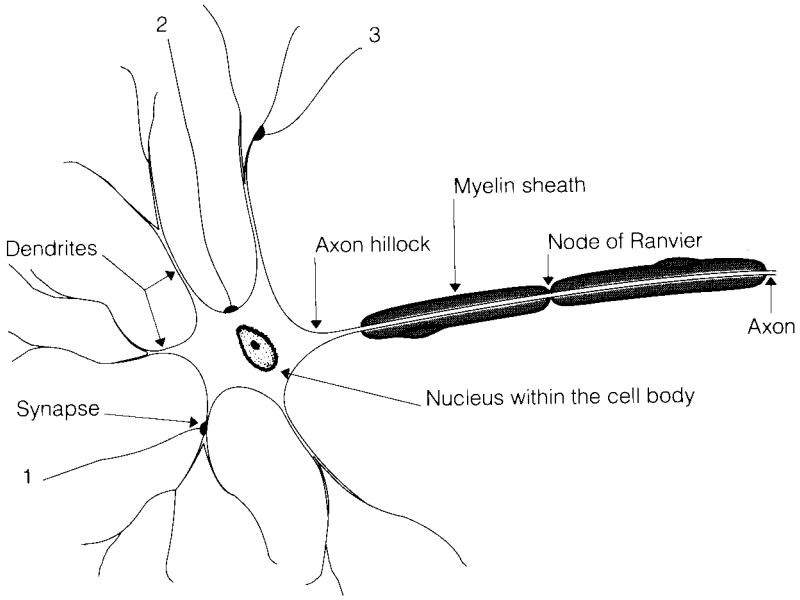
SA og grafpartisjonering I tillegg til å være en grunnleggende artikkel om simulert størkning generelt, er (Kirkpatrick, 1984) et av arbeidene som bruker teknikken til grafpartisjonering, selv om den ikke er så grundig når det gjelder det.

En artikkel som derimot er grundig (Johnson, Aragon, McGeoch, & Schevon, 1989), konkluderer med at det er viktig å finjustere parametrene i SA, og hvis man gjør det, kan man oppnå svært gode resultater. Ulempen er at algoritmen bruker lang tid hvis man skal få gode svar.

Nevrale nettverk

Dette er en metode som har fått et veldig oppsving etter at noen fant det bevingede navnet "nevralt nettverk" som en beskrivelse. Navnet er inspirert av måten beregninger foregår i hjernen.

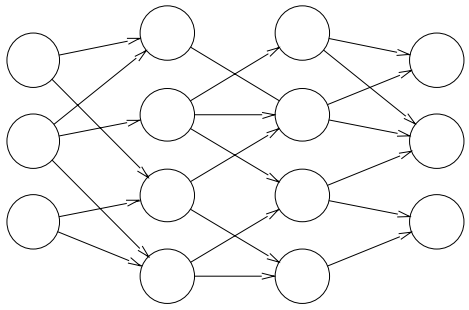
Nevrale nettverk i hjernen Den viktigste bestanddelen er nerveceller, eller nevroner, som er koblet sammen i et nettverk. Figur 2.14 viser et nevron.



Figur 2.14: Skisse av et nevron.

Hvis det kommer nok (positive) signaler inn på dendrittene, vil nevronet "fyre av", og sende et signal videre ut på axonet. Ett nevron vil naturligvis ikke være så veldig interessant, men når en menneskehjerne har av størrelsesorden 10^{11} nevroner blir det straks flere muligheter.

Nevrale nettverk i en datamaskin I matematikk- og informatikksammenheng er nevrale nettverk rettede grafer som ofte er ordnet med noen inn-noder som er koblet til endel beregningsnoder, som igjen er koblet til ut-nodene. Figur 2.15 er et eksempel på en vanlig type nevralt nettverk, et lagdelt "feedforward"-nett.



Figur 2.15: Et mulig nevralt nettverk.

Hver enkelt kant har en tilhørende vekt. Hvis en node får nok (positive) signaler inn til at det overstiger en gitt grense (etter at signalet har blitt justert ut i fra vektingsverdien), vil noden sende et signal videre på sine utgående kanter.

Det interessante med dette er at man til en viss grad kan få nettverket til å lære. Læringen foregår ved at man lar inn-nodene ha visse verdier, og justerer vektene utifra om ut-nodene får de ønskede verdiene. For lagdelte "feedforward"-nett kan man benytte den såkalte "backpropagation"-algoritmen til denne justeringen. Hvis man justerer vektene til å gi riktig resultat for testdataene man har, vil man forhåpentligvis få de forventede ut-verdiene også på ukjente inn-verdier. For mer om dette, se en innføringsbok om nevrale nettverk, f.eks (Hertz, Krogh, & Palmer, 1991).

Nevrale nettverk og grafpartisjonering Tilsvarende som simulert størkning, er nevrale nett en metode som kan anvendes på mange forskjellige problemer, med varierende grad av suksess. (Hérault & Niez, 1989) har prøvd å benytte den til graf k-partisjonering, og sammenligninger med simulert størkning. Det ser for meg ut som om denne artikkelen mest er ment for å vise at nevrale nettverk kan brukes på dette feltet også, og ikke så mye for å konkurrere med de beste andre metodene som finnes.

I (Rao & Patnaik, 1989) blir en nevral nettverksalgoritme benyttet til å lage en starttilstand for KL. Artikkelen konsentrerer seg om plassering av standard celler, men den samme fremgangsmåten kunne man tenke seg brukt til grafpartisjonering også. Det nevrale nettverket forbedrer en initiell grafoppdeling, men resultatet er robust mot endringer i den initielle grafen. I artikkelen er det oppnådd gode resultater, men det er bare brukt ganske små probleminstanser i eksemplene, så man kan frykte at algoritmen er tidskrevende.

Genetiske algoritmer

En annen fremgangsmåte som er hentet fra biologien, er genetiske algoritmer (GA). Det som simuleres er måten "kampen for å overleve" gjør at det er de beste arveanleggene som blir overlevert gjennom generasjonene. Man har derfor forskjellige genetiske operatorer, f.eks. parring (som tar to løsninger og setter dem sammen til en ny løsning) og mutasjon (som gjør en eller flere tilfeldige endringer i en løsning). Så setter man i gang en simulering med forskjellige individer (løsninger), og man lar de beste (bestemt av en evalueringsfunksjon) kombineres til nye, osv.

Dette er en generell metode som kan brukes til å løse mange kombinatoriske problemer. For å benytte den til å løse et spesifikt problem må man finne ut hvordan man skal definere en sammensetning av to løsninger, i hvor stor grad man skal se på løsninger som ikke er optimale og hvor lenge man skal holde på før man sier seg fornøyd med en løsning.

Hvis en genetisk algoritme er egnet for parallellisering kalles den for PGA. En slik PGA er (Laszewsski & Mühlenbein, 1991). De får gode resultater i forhold til 2 andre PGAer, men de har ikke tatt med noen sammenligning med andre typer algoritmer. Ofte vil (P)GAer havne i omtrent samme klasse som algoritmer for simulert størkning. Ihvertfall er det mye likt i hvordan algoritmene bruker randomisering for å styre søket og komme vekk fra lokale minima. Det betyr at begge metodene gir opphav til gode resultater, men bruker lang tid på å finne dem.

En GA for partisjonering som oppnår gode resultater og som samtidig er relativt rask, er beskrevet i (Bui & Moon, 1994). De bruker forholdskutt som kriterium. Isteden for å bare basere seg på at tilfeldige mutasjoner og parringer skal forbedre løsningene, blir det også brukt en forenklet variant av Fiduccia og Mattheyses' algoritme for raskt å forbedre løsninger. De gjør også noe preprosessering for at de genetiske operatorene skal fungere bedre. De får gode resultater på testdata sammenlignet med andre nylig publiserte metoder.

Egenverdi-metoder

En metode som har visst gode resultater er de som kalles spektrale metoder, eller egenvektor metoder. Jeg skal ikke gå inn på alle detaljene her, bare skissere kort hva det dreier seg om. En representasjon av en graf er som en 2 dimensjonal nabomatrise, C. I rute (i, j), ligger det 0 hvis det ikke går noen kant mellom node i og node j, og 1 — eller vekten på kanten, c_{ij} — hvis det går en kant der.

La x_i, y_i være koordinatene til node nummer i. For å finne disse, kan vi formulere en kostnadsfunksjon som måler summen av kvadratet til lederlengdene for 2-terminal nett:

$$L(x, y) = \frac{1}{2} \sum_{i,j=1}^{n} c_{ij}[(x_i - x_j)^2 + (y_i - y_j)^2] = x^T Bx + y^T By$$

hvor $B = [b_{ij}]$ er en modifisert nabomatrise definert som

$$B = D - C$$

der D er den diagonale matrisen med

$$d_{ij} = \sum_{j=1}^{n} c_{ij}.$$

I tillegg må en løsning tilfredstille noen krav som innebærer at modulene har lovlige posisjoner (f.eks at de ikke overlapper). Dette kan settes opp som n ligninger som må tilfredstilles.

Egenvektor-metoden for plassering ble introdusert i (Hall, 1970) og er ekvivalent med å minimere funksjonen $x^T Bx$ ved å ta hensyn til to av de n kravene til at modulene skal ha lovlige posisjoner. Ved å finne de to minste egenvektoren til B og de tilhørende egenverdiene, finner man x-koordiatene til modulene i planet. Siden man ikke tar hensyn til alle kravene for å få lovlige posisjoner, må man justere posisjonene til slutt, slik at man skal få en lovlig løsning.

I (Hagen & Kahng, 1992a) blir dette isteden brukt til partisjonering med forholdskutt som kriterium, som gir en algoritme som oppnår gode resultater. En variant av algoritmen arbeider på "intersection" grafen til nettliste-grafen. (Man lager en intersection-graf ved å la kanter tilsvare noder, og omvendt.) Utifra denne følger de omtrent samme metode for å finne en oppdeling.

Algoritmene har følgende egenskaper: De er relativt raske, har en angitt nedre grense for feil, stabil (forutsigbar) ytelse — trenger ikke "ta beste av flere tilfeldige starttilstander", og de skalerer godt.

Det er også andre som har forsøkt å knytte egenvektorer til grafpartisjonering (Donath & Hoffman, 1973; Barnes, 1982), men som har beregner egenvektorene til andre matriser enn B. Disse har imidlertid ikke fått like gode resultater.

(Riess, Doll, & Johannes, 1994) oppnår gode resultater med en lignende metode, men som bruker en lineær kostnadsfunksjon, dvs. en funksjon som måler summen av lederlengdene istedenfor å måle summen til kvadratet av lederlengdene.

En ulempe med alle algoritmene som bygger på spektrale metoder, er ta de forutsetter at det er vanlig graf som skal partisjoneres. En hypergraf kan ikke representeres i en nabomatrise. For benytte disse algoritmene på praktiske VLSI-problemer må man derfor simulere hypergrafer med vanlige grafer. Dette blir nærmere gjennomgått i avsnitt 2.3.4.

Parallelliserbarhet

I tillegg til de andre algoritmene som egner seg for parallellisering som er nevnt tidligere, (Talbi & Bessière, 1991; Laszewsski & Mühlenbein, 1991; Meuenier, 1989), er det introduseret en iterativ algoritme for bipartisjonering med min-kutt-kriteriet som egner seg godt for parallellisering og for store grafer (Savage & Wloka, 1991). Den baserer seg på å bytte mange noder (en såkalt Mob) på en gang. En Mob blir valgt ut ved å plukke ut enkeltnoder som har mest å tjene på å skifte side.

Algoritmen har blitt implementert på parallellmaskinen CM-2 (Connection Machine-2), med 32K prosessorer. Forfatterne antar at algoritmen også vil være godt egnet for serielle superdatamaskiner med et nøye balansert minnehierarki, som f.eks en Cray, pga måten minnet aksesseres på.

Det blir også gjort sammenligninger med KL og SA, hvor det blir vist at de er henholdsvis P-komplette og P-harde. Dette er uttrykk som blir definert i artikkelen, og som innebærer at de ikke lar seg parallellisere spesielt godt. (Her bør det nevnes at P-komplett tidligere ble brukt om det som nå kalles NP-komplett, før terminologien på området var helt fastsatt.)

Andre algoritmer som er foreslått

Algoritmer som bruker maks-flyt-min-kutt-kriteriet (Ford & Fulkerson, 1962) beskriver en algoritme med orden $O(n^2)$ for å finne det kuttet som gir minste antall kanter kuttet, som ligger på veien mellom to spesifikke noder. Hvis man kjører denne algoritmen med alle nodene som startpunkt, og for hvert startpunkt lar alle de andre nodene være sluttpunkt, vil man få en orden $O(n^4)$ algoritme som finner den oppdelingen av grafen som gir færrest kanter kuttet (dvs som finner den optimale løsningen utifra maks-flyt-min-kutt kriteriet).

Denne "naive" algoritmen kan forbedres hvis grafen lar seg dele opp i p komponenter som er "tri-connected" (dette beskrives nærmere i artikkelen). Da kan vi lage en algoritme med orden $O(n^4/p^3 + n^2)$, som vist i (Shing & Hu, 1986).

Selv om ordenen til disse algoritmene er polynomisk, så er den fremdeles så høy at det kan være uaktuellt å bruke den på store probleminstanser av den typen man kan støte på i VLSI. Uansett, siden oppdelingene normalt blir så skjeve at de er verdiløse, kan maks-flyt-min-kutt ikke brukes direkte.

Algoritmer som bruker separatorer (Bui et al., 1987) benytter separatorer som et element i en min-kutt-algoritme. Algoritmen vil ofte komme med optimalt svar, og hvis den ikke finner det optimale svaret, kommer den ikke med noe svar i det hele tatt. De viser at min-kutt problemet er NP-komplett også for den spesielle graftypen.

(Liu, 1989) er et eksempel på en artikkel som fjerner noder (ikke kanter) for å dele opp grafen. Bruker forbedringsveier (en vanlig teknikk for å løse matching-problemer) for å forbedre en løsning. Dette er ikke spesielt relevant for VLSI direkte, men metoden kan eventuelt benyttes som en delrutine.

(Leiserson, 1980) presenterer en algoritme for å lage gode utlegg for klasser av grafer som har gode separator-teoremer. Dette er trolig lite interessant for generelle nettlister, siden de ikke kommer inn under dette.

Algoritme som bipartisjonerer ved å først finne grupper I (Shin & Kim, 1993) blir det beskrevet en algoritme for å foreta bipartisjonering med min-kutt-kriteriet. Den bruker grup-

pering (som blir beskrevet senere) for å få mindre grafer å arbeide med underveis, a la (Bui et al., 1989). Deres kriterium for å lage grupper er som følger:

Nærheten av to noder eller grupper, C og D er gitt ved følgende formel:

$$Nærhet(C,D) = \frac{num\_cnet(C,D)}{\text{MIN}(num\_pin(C),(num\_pin(D)))} - \alpha \times (cl\_size(C,D)/avg\_size)$$

Hvor

$num\_cnet(C,D)$ er antall felles nett mellom C og D
$num\_pin(C)$ er antall 'pins' i C
$cl\_size(C,D)$ er størrelsen på den nye gruppen som oppstår hvis C og D blir satt sammen
avg_size er gjennomsnittlig størrelse på gruppene
α er en konstant som i deres implementasjon blir satt til $0.5/200$ for standard celle og gate-array utlegg.

Den første faktoren representerer tiltrekningskraften mellom gruppene/nodene, mens den andre faktoren representerer den frastøtende kraften som sørger for at gruppene får tilsvarende størrelse.

Disse gruppene som nå er laget blir så partisjonert i to deler for å finne et godt startpunkt for den endelige partisjoneringen. Resultatene er gjennomsnittlig 60% bedre enn Fiduccia og Mattheyses algoritme målt i antall nett som krysser.

2.3.4 Utvidelse til hypergrafer

Til nå har jeg sett på grafpartisjonering, men i VLSI-sammenheng er det nettlister, ikke grafer, som skal deles opp. I en graf går hver kant mellom nøyaktig to noder. I en nettliste går det som tilsvarer kanter, altså nett, mellom to eller flere terminaler (tilsvarende noder). En slik generell graf som en nettliste er et eksempel på, kalles for en hypergraf, og kantene kalles hyperkanter.

Kostnaden til å kutte hyperkanter i to deler

Hvis kostnaden på kutte en hyperkant som forbinder to noder (altså en vanlig kant) er 1, hva skal da kostnaden være med å kutte en hyperkant som forbinder 10 noder? Hvis man sørger for at alle nodene på hver sin side av kuttpunktet er forbundet innbyrdes, trenger man ikke å la nettet bli kuttet mer enn én gang. Derfor er det vanlig å la kostnaden på å kutte en hyperkant også være 1.

Utvide algoritmene til å fungere for hypergrafer

Mange algoritmer kan konverteres til å fungere også for hypergrafer ved å utvide datastrukturen til å ta hensyn til at kanter kan gå mellom mer enn mer enn to noder. Selve logikken i algoritmen blir ikke så veldig forskjellig i mange tilfeller.

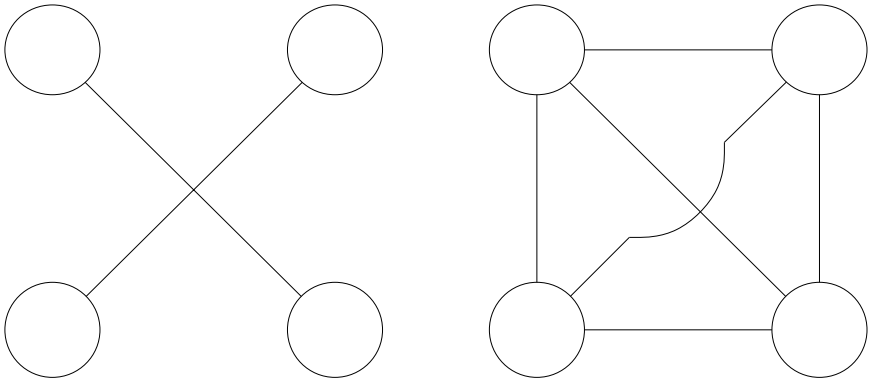
I artikkelen (Schweikert & Kernighan, 1972) blir det f.eks diskutert hvordan man kan modifisere KL til å fungere for hypergrafer. Den modifiserte metoden kalles for den generaliserte Kernighan og Lin-algoritmen (da gir den i tillegg mulighet til å angi at bestemte noder skal være faste, slik at de ikke kan ombyttes).

Tilsvarende er det mulig å gjøre for mange andre algoritmer, men det er noen problemer. I noen tilfeller vil et hopp fra grafer til hypergrafer innebære vesentlig høyere orden på algoritmen. Ofte vil det kunne bety et skille fra at problemet er løsbart i polynomisk tid, til at det er NP-komplett.

I andre tilfeller er det umulig å beholde logikken essensielt uendret. Dette gjelder spesifikt de algoritmene som baserer seg på å beregne egenverdier av nabomatrisene. Siden en hypergraf ikke kan representeres av en nabomatrise, er det ikke mulig å oppdatere algoritmen. For algoritmer som dette, er eneste mulighet å finne en vanlig graf som fungerer som en tilnærming til hypergrafen vi ønsker å partisjonere, og så arbeide på den. Dette vil ikke kunne gi optimale svar, i hvertfall ikke uten å kombinere det med en algoritme som arbeider direkte på en hypergraf.

Modelere hypergrafer med vanlige grafer

Hvis man ikke ønsker eller har mulighet til å befatte seg med hypergrafer direkte, kan man "simulere" en hypergraf med en vanlig graf. Den vanligste måten å gjøre dette på, er ved å la det gå en kant fra en node til en annen dersom de er forbundet med en hyperkant (Lengauer, 1990, avsnitt 6.1.5). Figur 2.16 viser forskjellen mellom fire noder som er forbundet med henholdsvis seks kanter og en hyperkant.

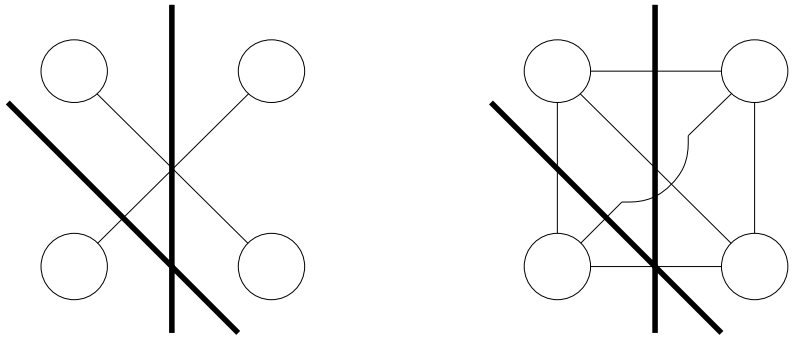


(a) En hypergraf med én hyperkant.

(b) En vanlig graf med seks kanter.

Figur 2.16: En kant i en hypergraf kan gå mellom mer enn to noder. I (a) går en hyperkant mellom fire noder.

For å se at en slik simulert hypergraf ikke vil oppføre seg på samme måte som en ekte hypergraf, kan vi se hva som skjer når vi kutter opp grafen i to deler.



(a) Én hyperkant blir kuttet uansett hvordan vi deler opp hypergrafen.

(b) Fire kanter blir kuttet hvis vi deler opp med to noder på hver side, men bare tre blir kuttet hvis man deler ujevnt.

Figur 2.17: Grafen skiller mellom de to måtene å dele på, mens hypergrafen ikke gjør det.

Som vi ser av figur 2.17 blir det svært dyrt å kutte en hyperkant etter at den er konvertert til mange vanlige kanter. Kuttverdien øker kvadratisk med antall noder som hyperkanten forbinder. For å motvirke dette kan man legge vekter på kantene for å redusere kostnaden ved å dele slik kanter. En vanlig måte å sette vekter på, er å konvertere en hyperkant som går mellom k noder til en komplett graf for de k nodene med en vekt på $1/(k-1)$ per kant. Det er foreslått flere andre forslag til hva man bør sette vektene til uten at jeg skal gå inn på dette her.

Uansett hvordan vi setter vektene på kantene vil grafen ikke være en nøyaktig modell av hypergrafen. Resultatet av å dele opp grafen blir forskjellig avhengig av om man deler jevnt og ujevnt, mens det ikke er noen forskjell på å dele opp hypergrafen jevnt eller ujevnt. Derfor vil det ikke være mulig å simulere en hypergraf helt korrekt med vanlige grafer.

Det hjelper heller ikke å sette ulik vekt på forskjellige kanter (inkludert å sette vekten til 0 på noen kanter, som tilsvarer å fjerne dem), siden dette nødvendigvis vil måtte bli asymmetrisk (to noder i den simulerte hyperkanten er forbundet sterkere/svakere enn andre). En hyperkant er ikke asymmetrisk, derfor vil en asymmetrisk modell ikke kunne være helt riktig.

2.3.5 Problemformulering for utvidelse til multipartisjonering/gruppering

En annen grunn til at problemformuleringene til nå ikke passer helt med det vi ønsker, er at vi ønsker å dele opp i mer enn to deler. Det vi har snakket om til nå kalles bipartisjonering, mens å dele opp i mer enn to kalles multipartisjonering. Hvis antall partisjoner (gjerne kalt k) er bestemt på forhånd, kalles det k-partisjonering. Hvis vi skal dele opp i mange deler (mer enn ca 10), eller antall deler ikke er bestemt på forhånd, kalles det ofte gruppering (eng. *clustering*). Her er det litt forskjellig bruk uttrykkene, fordi noen bruker gruppering hvis vi fokuserer på å maksimere antall kanter som går innen delene, som beskrevet i avsnitt 2.3.5.

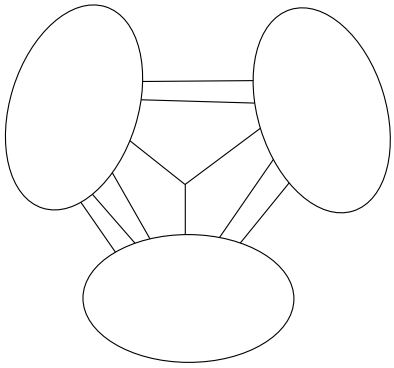
Jeg har ikke sett noen som har brukt multipartisjonering om partisjonering der man fjerner noder, bare om partisjonering der man fjerner kanter. Man kan godt formulere et slikt nodeseparator-kriterie, men det er tvilsomt om det svarer til problemer man støter på i praksis.

Når man går fra å dele i to til å dele i mer enn to, må man finne en formulering av problemet som passer med det vi faktisk ønsker. En utvidelse av problemet til multipartisjonering gir nemlig mange valgmuligheter også med hensyn til hva som skal vektlegges når vi skal se hvor god en oppdeling er. En bestemt måte å avgjøre dette på kalles en kriteriefunksjon (eng. *objective function*) eller kostnadsfunksjon (eng. *cost function*).

Først skal vi se på en faktor som vi må håndtere hvis vi skal dele hypergrafer, men som ikke kommer inn ved deling av vanlige grafer.

Kostnaden ved å kutte hyperkanter i flere deler

Når man todeler, vil en hyperkant enten gå mellom de to delene eller ikke. Ved deling i tre eller mer, er det flere muligheter, som demonstrert i figur 2.18.



Figur 2.18: En hypergraf med ukjent antall noder delt i tre deler. Hvor mye skal en hyperkant mellom fler enn to partisjoner koste?

En hyperkant kan holde seg innenfor en del, den kan gå mellom to deler, eller mellom flere deler. En presis definisjon av hypergraf k-partisjoneringsproblemet må bestemme kostnaden ved å dele en hyperkant på de forskjellige måtene. Hva man skal bestemme som kostnad, er avhengig av hvilket problem man egentlig ønsker å løse. Her kommer noen forslag på mulige kostfunksjoner som er diskutert i litteraturen (Sanchis, 1989), sammen med et eksempel på anvendelser:

- Et nett som har noder i nøyaktig k deler har kostnad $k - 1$.

  Dette målet vil være relevant for mange delproblemer omkring utlegg av VLSI.

- Kostnadsfunksjonen teller antall nett som har noder i mer enn en del. Det tilsvarer at kostnaden ved å kutte et nett er

$$\text{Kostnad} = \begin{cases} 0 & \text{hvis } k = 1 \\ 1 & \text{ellers} \end{cases}$$

  Dette kostnadsmålet kan være brukbart i en sammenheng hvor delene er prosessorer, nodene er beregninger som skal utføres, og nettene er data som skal aksesseres av flere beregninger. Ved å måle hvor mange nett som har noder i mer enn en del, finner vi hvor mange enheter med felles minne (eng. *shared memory*) vi trenger totalt.

- (Krishnamurthy, 1987) omhandler et kriterium der et nett som har noder i nøyaktig k deler koster $k(k-1)/2$. Dette tilsvarer at den vanlige modellen for å simulere hyperkanter med vanlige kanter, som beskrevet i avsnitt 2.3.4, er representativ.

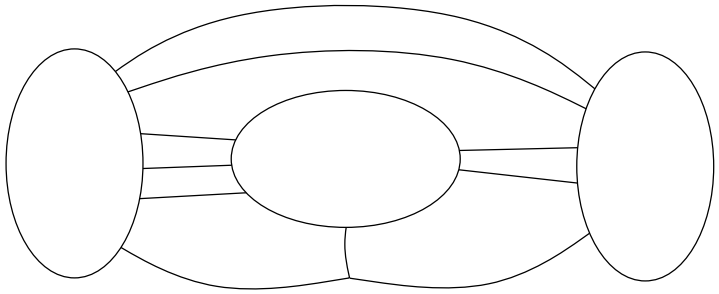
  Dette siste målet er egnet i distribuerte systemer hvor det er ønskelig å tilordne en enhetskostnad for hver prosessor-til-prosessor kommunikasjon.

Kostnadsmål hvor delene har posisjoner

Til nå har vi sett at tilstedeværelsen av hyperkanter vil komplisere generaliseringen av bipartisjonering. Det er en annen faktor som også kommer inn ved k-partisjonering som ikke er avhengig av hyperkanter, og det er hvorvidt det er like "langt" mellom alle partisjonene.

I noen anvendelser er det svært drastisk om en node havner i en del eller en annen del. Slik er det ikke nødvendigvis i VLSI. Her er grunnen til at vi deler opp at vi til slutt ønsker å plassere nodene. Det er ikke noen vesensforskjell på om to noder havner i samme del, eller i forskjellige deler. Hvis delene blir plassert tett inntil hverandre, kan det hende at to noder som er plassert i nabodeler til slutt havner nærmere hverandre enn to noder som er plassert i samme del.

Når vi har tre deler (eller fler), er det ikke sikkert at det koster like mye å kutte en kant mellom to deler, som mellom to andre deler. Dette kan f.eks komme av at delene har posisjoner, og at kostnaden på nettene er avhengig av avstanden mellom delene. Ved å tegne den samme konfigurasjonen som i figur 2.18 på en annen måte, kan vi illustrere dette poenget. Det er gjort i figur 2.19.

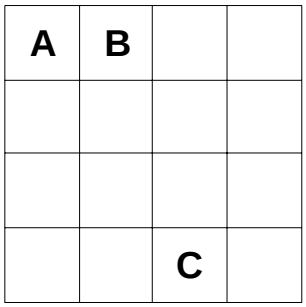


Figur 2.19: Den samme hypergrafen fra figur 2.18 partisjonert på samme måte, men tegnet forskjellig. Gjør avstanden mellom delene at noen kuttede kanter koster mer enn andre?

I denne situasjonen er ikke en kant lik enhver annen. Det "koster" mer å legge en lang leder enn en kort. Det er vanskelig å måle nøyaktig hvor mye mer det koster, men at det har en høyere pris er klart.

Det vi egentlig ønsker er å finne en oppdeling som vil være enkel å rute innenfor et lite areal. Dette har med å minimere gjennomsnittlig lederlengde å gjøre. Hvis det bare er ett nett som skal rutes, er dette problemet med å finne det minste rektilineære Steiner-treet som forbinder nodene, som beskrevet i avsnitt 2.5.2. Formulert som et desisjonsproblem er dette NP-komplett. Siden nettene kan gå i veien for hverandre, er det enda vanskeligere å beregne dette presist.

Ofte vil vi ha en forhåndsdefinert struktur vi ønsker å dele opp i, og ofte vil det være et rutemønster, som i figur 2.20.



Figur 2.20: Antall kanter kuttet er ikke tilstrekkelig som kriterium ved k-partisjonering. Det er kortere fra A til B, enn fra A til C.

På dette punktet begynner vi å bevege oss over i plasseringsfasen. Hvorvidt vi skal holde partisjonering og plassering helt adskilt, eller om vi skal prøve å gjøre begge deler på en gang er usikkert. Dette behandles nærmere i avsnitt 3.7.4.

Maksimere antall kanter som går internt grupper

Når vi utvider til k-partisjonering, kan vi forsøke å minimere antall kanter som går mellom gruppene (som omtalt over), eller vi kan konsentrere oss om innmaten av gruppene og forsøke å maksimere antall kanter der. Dette tilsvarer den varianten av bipartisjonering som er beskrevet på side 21. Denne metoden vil naturlig lede til en algoritme som er "bottom-up", mens det å minimere antall kanter som går mellom gruppene ofte leder til "top-down"-algoritmer.

Optimale løsninger vil være optimale uansett hvilken variant man velger her. Men tilnærmet gode løsninger for den ene varianten behøver ikke være tilsvarende gode med den andre varianten.

Dette å konsentrere seg om innsiden av gruppene er egnet hvis man har et bestemt antall noder i hver gruppe (og dermed også et bestemt antall deler), eller hvis vi ikke bestemmer antall noder/grupper på forhånd. Hvis vi ikke vet nøyaktig hvor mange deler vi skal dele opp i vil vi typisk dele opp i ganske mange deler, og nøyaktig antall vil være avhengig av hvor tett kommunikasjon det er. Hvis kretsen består av fem komponenter som hører sterkt sammen, vil algoritmen returnere det. En annen krets kan kanskje deles opp i 17 deler. Dette varierer altså med instansene.

2.3.6 En gjennomgang av forskjellige kriterier

Det optimale kriteriet for en god oppdeling er at det gir et godt utlegg. Dette blir imidlertid for vagt. Da må vi gjøre et helt utlegg for å se om vi fikk en god partisjonering. Vi forsøker istedenfor å konkretisere kravene litt mer på et tidligere tidspunkt i prosessen.

Gitt en nettliste $G = (V, E)$ med n moduler i $V = \{v_1, v_2, \ldots, v_n\}$, lag en k-veis oppdeling, P^k, som deler V i k disjunkte grupper $C_1, C_2, \ldots, C_k$, som minimaliserer en gitt kostnadsfunksjon $f(P^k)$.

Disse arbeidene er mest relevante hvis k er mye mindre enn antall noder, f.eks $k <= 10$ for $n >= 1000$. Nedenfor blir forskjellige kostnadsfunksjoner gjennomgått.

Minimum gruppeforhold (eng. *cluster ratio*)

Minimer

$$f(P^k) = \frac{c(P^k)}{\sum_{i=1}^{k} \sum_{j=i+1}^{k} |C_i| \times |C_j|}$$

hvor $c(P^k)$ er antall nett som går mellom to eller flere grupper i P^k.

Dette er en generalisering av forholdskutt (og av den grunn NP-komplett) som er foreslått i (Yeh, Cheng, & Lin, 1992). Artikkelen foreslår også en tilhørende probabilistisk algoritme "korteste vei gruppering" (eng. *shortest-path clustering*) som går ut på å fjerne de korteste veiene mellom tilfeldige noder iterativt. Prosessen stopper når det er k komponenter som ikke har forbindelse til hverandre. Algoritmen vil probabilistisk fange forholdet mellom maks-flyt-min-kutt og forholdskutt.

Minimum skalert kostnad (eng. *scaled cost*)

Minimer

$$f(P^k) = \frac{1}{n(k-1)} \sum_{i=1}^{k-1} \frac{E_i}{|C_i|}$$

Dette ble foreslått i (Chan, Schlag, & Zien, 1994), og er en annen måte å generalisere forholdskutt til k-partisjonering (og er også NP-komplett).

Forskjellige mål på hva som er gode grupper

Man kan også gi kriterier for hvor god en gruppe er. Et slikt kriterium vil imidlertid bare si noe lokalt om en enkelt gruppe.

k-l-koblinger.

I (Garbers, Promel, & Steger, 1990) blir det presentert en måte å finne ut om to noder bør ligge i samme gruppe: To noder er k-l forbundet hvis og bare hvis det finnes k veier som forbinder dem slik at hver vei har en lengde på maksimalt l, og slik at ingen av veiene har noen kanter felles. (Parameteren k har altså ikke noe med antall grupper som blir laget å gjøre, det vil variere fra en krets til en annen.) Ideen er at hvis to noder er forbundet av mange korte veier, så er de sterkt forbundet. Gruppene blir definert ut i fra den transitive tilukningen av k-l-relasjonen.

Nøyaktig hvilke verdier som blir valgt for k og l har mye å si for hvor gode løsninger som blir funnet. Typisk vil det bli laget flere store grupper, og noen små (gjerne med bare en node).

Grad/separasjon (eng. *degree/separation*)

(Hagen & Kahng, 1992b) kommer med dette kriteriet for hvor god en gruppering er: graden til en gruppe er gjennomsnittlig antall nett som er inntil hver komponent i gruppen. Separasjonen er gjennomsnittlig lengde for korteste vei mellom to komponenter i gruppen. Grupper som har høy D/S verdi har høy kvalitet. Det er et godt mål, siden det tar med global forbindelsesinformasjon, men det er tungt å beregne hvis gruppene er store. Beregningen av separasjon krever orden $O(n^3)$ hvor n er antall komponenter i gruppen.

Gruppetetthet (eng. *cluster density*)

Gitt en gruppe med c noder vil gruppetettheten være $\frac{E}{M_c}$ hvor $M_c = \binom{c}{2}$ og E er total vekt på kantene i gruppen. Grupper med høyere tetthet har høyere kvalitet. Dette er et enkelt og intuitivt kriterium. Det vil favorisere små grupper siden verdien av M_c øker fort når c øker. Denne er omtalt i (Cong & Smith, 1993).

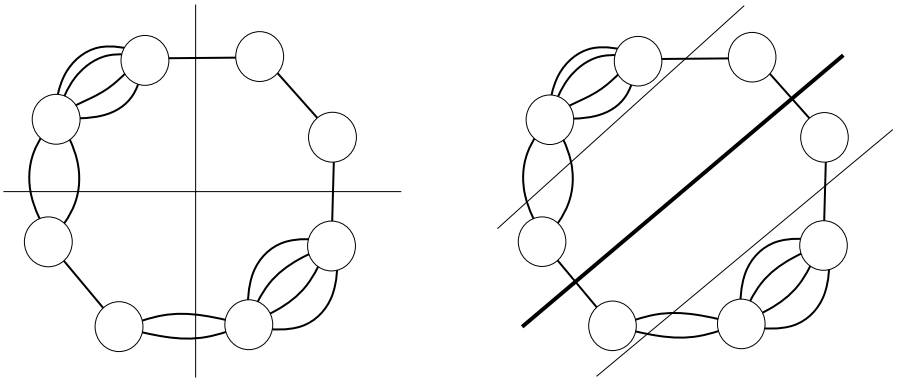
k grupper, uten begrensning på størrelsen

(Saran & Vazirani, 1995; Goldschmidt & Hochbaum, 1994) ser på problemet med å dele en graf i k uavhengige komponenter ved å fjerne kanter. Det er antall kanter som blir forsøkt minimert. Dette er altså en generalisering av maks-flyt min-kutt fra 2 til k. Hvis k er bestemt på forhånd er problemet løsbart i polynomisk tid (men med en høy orden på algoritmene). Hvis man vil ha den oppdelingen som kutter færrest kanter uavhengig av antall grupper som blir laget, er problemet NP-komplett (som vist i (Goldschmidt & Hochbaum, 1994)).

2.3.7 Algoritmer for multipartisjonering/gruppering

Simulere k-partisjonering med gjentatt bipartisjonering

Hvis man har en algoritme for å dele i to, og vi ønsker å dele i fire, er det nærliggende å først kutte i to, og så kutte i to igjen. Som figur 2.21 viser, vil dette ikke helt tilsvare å dele i fire med en gang. Hvis man deler opp i flere omganger, vil den første delingen ikke ta hensyn til at vi skal dele igjen. Dette er som oftest ikke det vi ønsker, så hvis man velger å bruke denne metoden må man regne med å få et dårligere resultat enn ved å dele direkte.



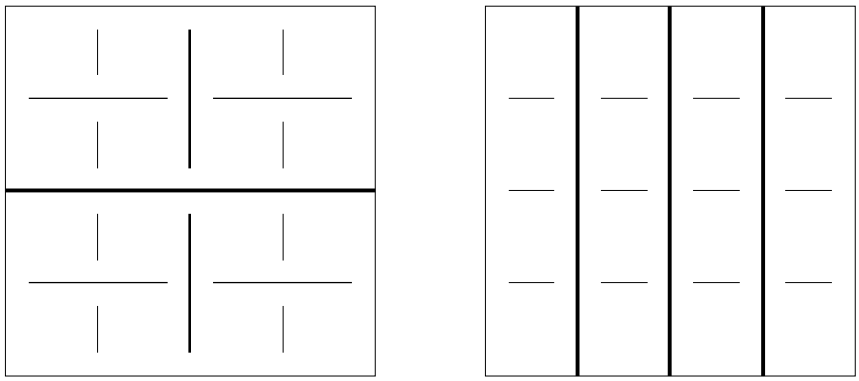
(a) Resultatet etter å ha foretatt en optimal oppdeling i fire. Dette gir seks kuttede kanter.

(b) Optimal oppdeling i to, etterfulgt av optimal 2-deling av hver av del. Gir totalt åtte kuttede kanter.

Figur 2.21: Gjentatt bipartisjonering gir dårligere resultat enn k-partisjonering direkte.

Metoden egner seg heller ikke helt godt hvis man skal del i f.eks tre deler. Til tross for disse problemene, er metoden likevel mye brukt, f.eks i (Dunlop & Kernighan, 1985). Det kommer av at den er grei å implementere, og at den gir gode nok resultater til mange anvendelser.

Hvis man skal bruke bipartisjonering gjentatte ganger, blir måten man organiserer rekkefølgen på oppdelingene viktig. Breuer behandler dette i (Breuer, 1977a, 1977b). Han har foreslått blant annet de oppdelingene som er vist i figur 2.22.



(a) Alternerende horisontale og vertikale kutt. En slik oppdeling minimerer behovet for forbindelser i midten av utleggsarealet.

(b) Ved å foreta alle vertikale kutt før alle horisontale, får man en oppdeling som kan egne seg dersom det er mange eksterne forbindelser.

Figur 2.22: Noen mulige måter å organisere oppdelingene på ved gjentatt bipartisjonering.

Min-kutt Kernighan og Lin foreslo også en måte å bruke deres algoritme til å fungere for k-partisjonering. Man begynner med en vilkårlig k-veis oppdeling. Denne blir så forbedret ved å kjøre KL på par av deler. Dette fortsetter helt til man ikke lenger får noen forbedring, eller til forbedringen er så liten i forhold til tidsforbruket, at vi ønsker å avbryte. Uansett vil de største forbedringene komme tidlig i prosessen. Denne meta-heuristikken er også anvendbar for andre iterative bipartisjoneringsalgoritmer.

Utvide algoritmene til å fungere for k-partisjonering

Algoritme med ytelsesgaranti Artikkelen (Feo & Khellaf, 1987) beskriver en tilnærmingsalgoritme for å løse varianten av grupperingsproblemet for vanlige grafer hvor vi ønsker like mange noder, B, i hver gruppe. Siden artikkelen er vanskelig fysisk tilgjengelig, har jeg bare lest beskrivelsen i (Lengauer, 1990, avsnitt 6.7). Algoritmen er enklest dersom antall noder i hver del er et partall, så det er det som blir beskrevet under.

Algoritmen fungerer ved å transformere instansen slik at det kan løses av et enklere problem, maksimal vekt matching (eng. *Maximum weight matching*), og ut i fra en optimal løsningen på dette problemet finner man en tilnærmet løsning på det egentlige problemet. Maksimal vekt matching arbeider på komplette grafer med vekter på nodene, så det første skrittet er å gjøre om den opprinnelige grafen slik at dette holder. Denne konverteringen innfører en ny kant med vekt 0 hvis det ikke var noen kant mellom to noder fra før, og beholder vektene på de eksisterende kantene. Hvis originalgrafen ikke har operert med vekter på kantene, setter vi vekten til 1. Vi kan også starte med en hypergraf ved å bruke metodene som er beskrevet i avsnitt 2.3.4 til å bestemme vektene på kantene.

Maksimal vekt matching finner en maksimal matching til grafen (dette problemet er beskrevet på side 27). Dette er greit, siden en komplett graf har kanter mellom alle nodene. Problemet består i å finne den maksimale matchingen som maksimerer summen av kantvektene i matchingen. Dette problemet kan løses i polynomisk tid, enten optimalt (Galil, 1986) eller litt raskere ved hjelp av gode heuristiske algoritmer (Avis, 1983).

Ved å slå sammen parene av noder som blir forbundet med kanter i matchingen til en gruppe, vil vi finne en optimal oppdeling i grupper med to noder i hver del. Hvis vi ber om flere noder i hver del, kan vi heuristisk redusere grupperingsproblemet til det enklere tilfellet. Vi velger $B/2$ kanter av gangen fra matchingen. Nodene som er inntil disse kantene slår vi sammen til en gruppe. Denne prosessen fortsetter til alle nodene har blitt tilordnet en gruppe.

Denne algoritmen har altså en egenskap som er svært ettertraktet: Den har en garanti for hvor dårlige løsninger den kan lage. Gitt at triangelulikheten holder for kantvektene (det er en rimelig forutsetning) vil vi ha følgende grense for feilen:

$$c_{opt} <= \frac{2(B-1)}{B} c(\mathrm{II})$$

Hvor

c_{opt} er den optimale løsningen, og
$c(II)$ er løsningen som algoritmen finner.

En sterkere grense enn dette kan vi ikke lage, for det finnes eksempler hvor denne feilen blir oppnådd. Jeg skal ikke gjengi hele beviset for at denne feilgrensen holder, men det bunner i at den maksimale matchingen inneholder en stor del av kantvektene i grafen.

Finne grupper med varierende størrelse (Garbers et al., 1990) finner grupper som oppfyller k-l-kriteriet. Artikkelen har med en konstruktiv algoritme som virker på vanlige grafer, og en variant som virker på direkte på hypergrafer.

For å gjøre resultatet mer anvendelig, blir eventuelle smågrupper klumpet sammen. Til dette brukes en naiv sammenklumping som kanskje kan forbedres. De sammenlignet sin algoritme med optimaliserte min-kutt-algoritmer, og fant at gruppering fungerer svært bra for sterkt strukturerte utlegg. For andre typer utlegg fungerer det ikke så bra, men kan da kanskje brukes som utgangspunkt for min-kutt.

Firedeling Firedeling (eng. *quadrisectioning*) er en metode for å dele en graf i fire deler direkte (Suaris & Gershon, 1989). Dermed unngår man problemet med vanlig bipartisjonering hvor det første kuttet ikke tar hensyn til det neste.

Hvis man skal dele opp i mer enn fire, vil man selvsagt havne i problemer igjen, men det er to faktorer som gjør at det allikevel er en stor forbedring:

- For det første trenger man færre kall på rutinen siden den deler opp i flere med en gang.
- Viktigere er det at denne fire-delingen ikke forskjellsbehandler noen av dimensjonene siden de blir behandlet samtidig. Derfor blir det en mye jevnere fordeling.

16-deling (Mayrhofer & Lauther, 1990) beskriver en metode som kan sees på som en utvidelse av firedeling til å dele i enda flere deler med en gang. Algoritmen bruker et kriterie som gir delene posisjoner, slik at dette kan sees på som en algoritme for plassering (og ikke partisjonering). For å finne ut hvor lett det er å rute utlegget, er det fint å ha et estimat over

hva lederlengden kommer til å bli når ruteren har koblet sammen alle nettene. Algoritmen benytter rektilineære Steiner-trær som mål på hvor god oppdelingen er (se avsnitt 2.5.2 for mer om Steiner-trær).

Steiner-trær representerer det korteste utlegget av et nett, gitt at det ikke er noen hindringer i veien. I interessante systemer vil det alltid være hindringer i veien, om ikke annet vil det ihvertfall være andre nett som konkurrerer om plassen. Derfor vil det ikke gi et helt korrekt estimat over lederlengden, selv om Steiner-trær benyttes.

Med det vil ihvertfall være mer korrekt enn den andre populære måten å estimere dette på: halve omkretsen (eng. *half perimeter*) av det omsluttende rektangelet til alle terminalene i nettet. Dette vil være omtrent riktig, og langt raskere å beregne enn Steiner-trær. En komplikasjon med å bruke rektilineære Steiner-trær som estimat, er nettopp at de er tunge å beregne.

For at beregningen av rektilineære Steiner-trær ikke skal ta for mye tid, blir de her beregnet på forhånd og lagret. Siden antallet Steiner-trær vokser fort (2^L hvor L er antall deler man ønsker å dele opp i), kan man ikke bruke denne metoden for særlig mer enn 16 deler. Hvis man ønsker å dele opp i mer enn det, må man bruke en tilnærmingsalgoritme for å beregne Steiner-trær, eller eventuelt kalle rutinen rekursivt. Begge disse metodene har uheldige egenskaper, men vil allikevel gi et mer nøyaktig mål på hvor god en oppdeling er enn ren bipartisjonering gjentatte ganger.

k-partisjonering Arbeidet (Sanchis, 1989) utvider (Krishnamurthy, 1984) fra bipartisjonering til k-partisjonering. Detaljene i dette arbeidet er nok ikke spesielt relevant i forhold til VLSI, pga den kostnadsfunksjon som hun arbeider med. Hun konsentrerer seg om den kostnadsfunksjon beskrevet på side 37 som egner seg best til å minimere felles minne.

Gruppering via konvertering til geometriske utlegg

I (Alpert & Kahng, 1993) blir det foreslått å beregne spektrale geometriske representasjoner (eng. *embeddings*) av nettlisten, og så bruke enkle geometriske partisjoneringsalgoritmer for å finne løsninger på grupperingsproblemet. Dette er et eget felt, som har utviklet forskjellige algoritmer som muligens er egnet. Konklusjonen er at dette kan brukes til VLSI, men at man ved å bare arbeide på en geometrisk representasjon vil miste verdifull informasjon fra nettlisten.

Det er vanskelig å si hvordan kriteriene for dette problemet henger sammen med kriteriene for det opprinnelige problemet.

Begrenset k-veis partisjonering (eng. *restricted k-way partitioning*)

Basert på erfaringene fra forrige avsnitt, foreslår (Alpert & Kahng, 1994b) et kriterium som kombinerer informasjon fra den geometriske representasjonen av nettlisten med informasjon direkte fra nettlisten.

De foreslår Restricted k-way partitioning (RP) med kriterier for hvordan nodene skal fordeles i gruppene og med nedre og øvre grenser for størrelser på gruppene.

Med disse kriteriene lager de en algoritme som er basert på at det er en sammenheng mellom løsninger av "Den Handelsreisendes Problem" (eng. *Traveling Salespersons Problem (TSP)*) og gruppering. Det finnes en algoritme for TSP som går via å løse et grupperingsproblem, og her blir et grupperingsproblem løst ved å se på en løsning av TSP. Løsningen

bruker romutfyllende kurver (eng. *spacefilling curves*), spesifikt Sierpinski-kurven. Dynamisk programmering blir utnyttet for å finne løsningene. Resultatet på det hele er en algoritme som gjør det bra for "minimum skalert kostnad"-kriteriet for $2 <= k <= 5$.

Noen andre algoritmer

En "random walk" algoritme er foreslått som bruker Grad/separasjon som kriterium (Hagen & Kahng, 1992b). En slik tilfeldig gange begynner i en node, og tar n^2 skritt rundt i grafen. Basert på sykler i nodesekvensene, bygger man opp grupper. Ulempen er at de har en orden på $O(n^3)$, hvor n er antall noder i grafen.

(Feo & Khellaf, 1987) passer for grafer, og gir ytelsesgaranti. Kriteriet som brukes bestemmer antall deler på forhånd, og antall noder i hver del. Den fungerer bra for relativt få noder i hver del.

(Alpert & Kahng, 1994a) ser på grupperingsmetoder som først sorterer nodene etter hverandre slik at sammenhengende delmengder av listen skal gi gode grupper. Deretter blir gruppene definert ved å splitte opp listen på passelige steder.

De lager så en generell algoritme for dette. De bygger opp en gruppe ved å stadig legge til nye noder. De ser bort fra "gamle" noder, ved at de har et vindu som glir over listen mens den bygges opp. Dette gir opphav på navnet på algoritmen, WINDOW. Avhengig av hvilke kriterier som brukes for å legge til de nye nodene, kan dette rammeverket simulere andre kriterier, f.eks bredde-først søk og dybde-først søk.

Algoritmer som fungerer direkte på hypergrafer

De fleste algoritmene som er nevnt her fungerer bare på vanlige grafer, og ikke på hypergrafer. Unntaket er k-l-algoritmen fra (Garbers et al., 1990). En annen algoritme som arbeider direkte med hypergrafer er beskrevet i (Lee, Chou, & Fu, 1993). Underveis i algoritmen blir nodene delt inn i to typer: De som er med i grupper, C, og de som ikke er det, G. C består initielt av k mengder, som hver består av en node hver. Disse nodene kalles frønoder, og er bestemt utifra en egen algoritme. Kvaliteten på frøene har mye å si for den endelige kvaliteten. Utifra studier av hvordan designeksperter velger ut frønoder, brukes følgende krav for å velge ut frø:

- Frøene må være uniformt distribuert utover hypergrafen.
- Det er lurt å velge ut noder som har forbindelser til mange andre noder.

Det blir beskrevet en algoritme som har disse egenskapene. Denne baserer seg på begrepet "nodedybde". For å velge ut frø blir nodene delt i k like store deler utifra dybden. Av disse blir de nodene som har flest forbindelser plukket ut. Her blir det sett bort fra store nett (større enn 5–11 noder), fordi de ofte er globale i den forstand at de forbinder mindre grupper.

For å gro gruppene velges en node og en gruppe, og så blir noden inkludert i gruppen. Dette gjentar seg til alle nodene har kommet med i en gruppe. Noden blir plukket ut på grunnlag av "inside-outside connectivity" (IOC), som er hvor mange kanter som er felles med noder i C og hvor mange som er felles med G og hvor mange som er felles med både C og G. Hvilken gruppe som noden skal slåes sammen med velges ut på grunnlag av hvor mange forbindelser det er til noden, og utifra hvor fulle gruppene er. Denne algoritmen kan enten brukes til generell k-partisjonering, eller til bipartisjonering.

2.3.8 Gruppering som tar hensyn til P- og N-transistorer

En annen viktig ting er at fordi vi opererer med CMOS teknologi, vil det være slik at den ene typen transistorer må være i brønner, mens den andre typen må være utenfor (eller i en annen type brønner) som beskrevet i avsnitt 1.2.4. En brønn må ha en viss størrelse, og transistorer kan ikke ligge for nær kanten av en brønn.

Dette fører til at man ønsker at hver enkelt gruppe som partisjoneringsprogrammet lager bare skal bestå av en type transistorer.

Den eneste artikkelen jeg har funnet som eksplisitt tar opp dette problemet i forbindelse med partisjonering er (Hughes et al., 1986). Der blir det beskrevet hvordan de legger høy kostnad på forbindelser mellom P- og N-transistorer, for at KL skal øke sjansen for å legge dem i samme gruppe. For å gå fra bipartisjonering til k-partisjonering, blir KL utført gjentatte ganger.

2.4 Plassering av minimoduler

Det er en mengde litteratur som omhandler floorplanning og plassering av moduler (Lengauer, 1990, kapittel 7). Siden vi i denne oppgaven ikke har mulighet til å påvirke høyde/bredde-forholdet på minimodulene, er det her bare snakk om plassering, og ikke om floorplanning. Siden plasseringsproblemet har en frihetsgrad mindre, er det et enklere problem å løse enn floorplanningsproblemet. Plasseringsproblemet i denne oppgaven er enda enklere enn vanlig, fordi alle modulene er omtrent like. Vanligvis består plasseringsproblemet av to deler:

- Et todimensjonalt pakkeproblem

  Pakkeproblemet innebærer hvordan moduler av forskjellig størrelse skal plasseres utover. Figur 1.9 på side 11 viser et eksempel hvor dette aspektet kommer inn.

- Et forbindelsesoptimaliseringsproblem

  Dette har med den delen av plassering som forsøker å finne en plassering som er lett å rute. Målet for hvor godt man klarer dette, er om utlegget lar seg rute med den plassen som er satt av. Hvis man alltid vil få utlegget til å gå opp ved å bruke litt ekstra plass, vil det være hvor mye ekstra plass som trengs som er målet for hvor god plasseringen er.

I dette tilfellet er alle minimodulene så godt som like, så her forsvinner pakkeaspektet. Mer om hvilke muligheter dette åpner for kommer i avsnitt 3.10.

2.5 Ruting mellom minimoduler

Ruting deles gjerne inn i tre typer. De to første har jeg nevnt tidligere: globalruting og lokalruting. Den siste måten er områderuting (eng. *area routing*) som rett og slett løser hele problemet på en gang. Dette har naturlig nok likheter med både globalruting og lokalruting, siden det løser begge delproblemene.

I denne oppgaven dukker alle tre typene ruting opp. Globalruting og lokalruting trengs for å finne sammenkoblinger mellom minimodulene, mens internt i minimodulene er det naturlig å bruke en områderuter. Fordi instansene er så små, vil en oppspalting av dette ikke være hensiktsmessig.

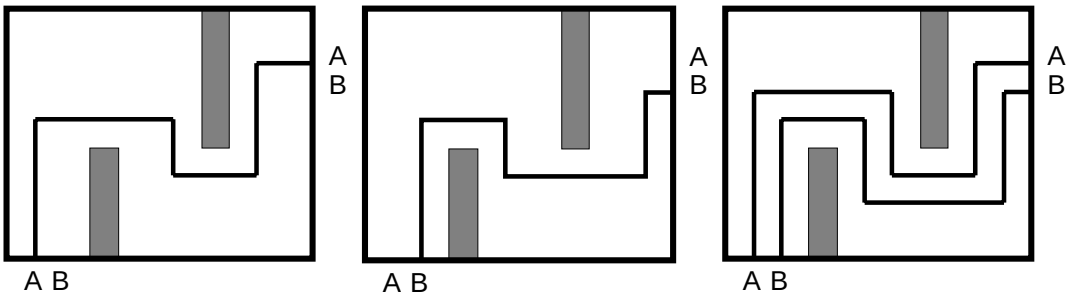
2.5.1 Teori rundt ruting

En enkel algoritme for å rute et utlegg er å gjenta følgende sekvens til alle nettene er ferdig rutet:

- Velg et nett som skal rutes
- Finn den korteste veien som forbinder nodene som hører til nettet

Det er flere problemer med denne enkle algoritmen. Et problem er at det å finne den korteste veien som forbinder et nett er vanskelig i seg selv. Hvis det bare er to noder i nettet, er det mulig å løse problemet i polynomisk tid. Men hvis det er mange noder i nettet, er problemet vanskeligere. Problemet er da identisk med å finne det minste rektilineære Steiner-treet (som beskrevet i avsnitt 2.5.2). Selv om man løser dette ved å bruke forskjellige heuristiske metoder, gjenstår det et stort problem med algoritmen som er skissert over.

Nettene som blir rutet tidlig har mye større sjanse til å komme fram enn de som blir rutet sent. De tidligere plasserte nettene vil være hindringer for de senere. Derfor er det viktig hvilke nett som skal rutes først. Men problemene er ikke over selv om man finner en god algoritme for denne deloppgaven. Det finnes nemlig konfigurasjoner som er slik at hvis det ene nettet rutes først langs den korteste veien vil det andre ikke kunne rutes, og omvendt. Hvis begge nettene får tilordnet suboptimale ruter, vil det gå bra. Et slikt eksempel er gitt i figur 2.23.



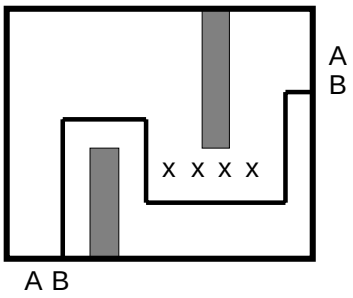
(a) Det ene nettet rutet først langs den korteste veien vil sperre for det andre...

(b) ... og omvendt.

(c) Bare hvis begge nettene får suboptimale ruter vil det gå bra.

Figur 2.23: Et eksempel på at ikke alle nettene kan rutes optimalt samtidig.

Det å rute alle nettene samtidig er vanskelig, så det er vanlig å rute nettene etter tur. En måte å håndtere dette på er å finne situasjoner hvor dette problemet vil oppstå, og å sørge for å sette av plass slik at senere nett lar seg rute, som illustrert i figur 2.24.



Figur 2.24: Ved å reservere noen punkter i matrisen for senere nett vil utlegget la seg rute.

Dette er implementert i en såkalt X-ruter (navnet kommer av X'ene som markerer reserverte punkter) i WIRES systemet (Namekawa, Suzuki, Takano, & Ohtsuki,). Siden artikkelen er på japansk, har jeg fulgt beskrivelsen og figurene fra (Kuh & Ohtsuki, 1990).

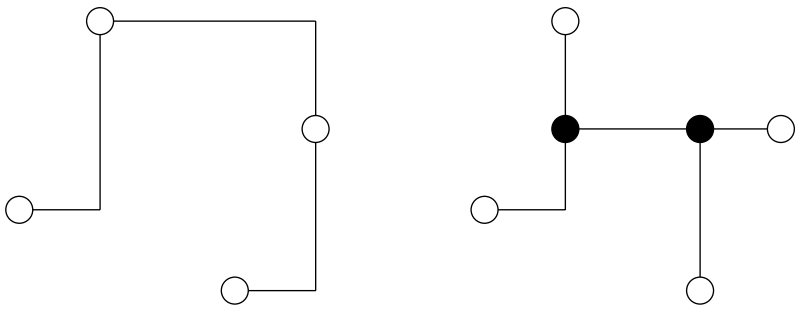
2.5.2 Minimum rektilineære Steiner-trær

Det å finne den korteste lederen som forbinder et antall terminaler, er ekvivalent med å finne det minimale rektilineære Steiner-treet som forbinder punktene. En måte å definere dette problemet på, som antar at minimum rektilineære spenntrær er kjent, er som følger:

Gitt en mengde P med n punkter, finn en mengde S med Steiner-punkter slik at det minimum rektilineære spenntreet over $P \cup S$ har minimum kostnad.

Et minimum rektilineært spenntre er det spenntreet som forbinder alle nodene i P, og som minimerer den totale Manhatten-avstanden.

Figur 2.25 viser forskjellen på disse to måtene å koble sammen punkter på.



(a) Minimum rektilineært spenntre.

(b) Minimum rektilineært Steiner-tre.

Figur 2.25: Minimum rektilineære spenntre og Steiner-tre for de samme punktene. De hvite punktene er de originale punktene P, mens de sorte punktene representerer mengden S av Steiner-punkter som er lagt til. Figuren er hentet fra (Griffith et al., 1994)

Steiner-tre-problemet er NP-komplett, men det er funnet gode tilnærmingsalgoritmer (Griffith et al., 1994). Disse finner løsninger som som hurtig kommer med løsninger som er garantert å være få prosent unna det optimale.

2.5.3 Globalruting mellom minimoduler

Globalruting blir ikke behandlet inngående i denne oppgaven. Jeg skal allikevel si noen ord om emnet. Det er foreslått mange algoritmer for globalruting (Kuh & Marek-Sadowska, 1986; Lengauer, 1990, kapittel 8). Rammeverket som brukes i denne oppgaven behøver ikke å påvirke globalrutingen i stor grad, så sannsynligvis kan en av disse brukes med godt resultat. I avsnitt 3.11.1 behandler jeg noen måter å tilpasse globalruteren til resten av systemet, som kanskje kan gi bedre resultater enn et helt "standard" opplegg.

2.5.4 Lokalruting mellom minimoduler

Tilsvarende som for globalruting, kan nok mange lokalrutere brukes direkte. Det trengs ihvertfall en koblingsboksruter til å ta seg av de mest generelle tilfellene. Sannsynligvis vil det også være hensiktsmessig å integrere en egen kanalruter. (LaPaugh & Pinter, 1989; Lengauer, 1990, kapittel 9) gir en oversikt over feltet kanalruting.

2.6 Detaljert utlegg av minimoduler

2.6.1 Teknologiuavhengighet

Det er i denne fasen at vi har mest med detaljene i designreglene å gjøre. Derfor er det på dette stedet vi må tenke på hvordan vi skal få et program som ikke blir for avhengig av bestemte designregler. Dette er beskrevet nærmere i avsnitt 3.4.4.

2.6.2 Plassering av transistorer

Gittermønster

Når man arbeider så detaljert som med å plassere transistorer, er det lett å bli overveldet av kompleksiteten hvis man skal ta hensyn til alle detaljer. Som nevnt, er det vanlig å bare operere med Manhattan-geometri. En annen vanlig forenkling, er å redusere oppløsningen vi arbeider med. Det gjør vi ved å plassere komponentene ut i et tenkt gittermønster (eng. *grid*). Det innebærer at det blir færre steder å sette transistorer, men også at datastrukturen blir mye enklere. Isteden for å arbeide med flyttall for å finne ut hva som ligger i nærheten, kan man bare sjekke noen få nabopunkter.

Dette er en svært vanlig forenkling. De eneste programmene som ikke bruker den, er noen kanalrutere og de fleste kompakteringsprogrammer. For kanalruternes vedkommende har man sterke føringer på hva som er lovlige instanser, og feltet er grundig forstått. Kompaktering skal jo nettopp gjøre små justeringer for å få utlegget litt mer kompakt, og da må man bruke alle relevante opplysninger.

Optimal plassering

De fleste artiklene som omhandler plassering av transistorer, beskriver heuristiske algoritmer for å finne tilnærmede løsninger på litt større problemer. Fordi de fleste systemer regner med å legge ut ganske mange elementer på en gang, er det en naturlig strategi. Mange systemer er regelbaserte, dvs at det heuristiske reglene er basert på hvordan utleggseksperter beskriver sin tenkemåte.

Et unntak er noen arbeider innen "line-of-diffusion" utleggsstilen, dvs der hvor man forsøker å la transistorene dele felles diffusjonsområde så mye som mulig — ved å la source noden fra den ene transistoren være koblet direkte til drain på den neste, får man et kompakt utlegg. Får å få dette til, må altså transistorene være ordnet i den rekkefølgen som gir lengst sekvenser med sammenhengende diffusjon. Til dette benyttes "Euler paths" og "dual trails".

Eksponentiell eksplosjon

I Arislands arbeid blir det foreslått å løse de oppdelte problemene optimalt. Det er svært vanskelig i det generelle tilfellet. Siden vi har sikret oss at probleminstansene alltid er relativt små, skulle det allikevel være mulig. Men antall kombinasjoner vokser fort, og er avhengig av (det innvendige) arealet (a) og antall transistorer (t) som skal plasseres. Det blir

$$\frac{a!}{(a-t)!} * t^4$$

Det første leddet måler antall mulige måter å plassere t transistorer i a posisjoner, mens det andre leddet juster for det faktum at hver transistor kan stå i fire forskjellige posisjoner.

Dette blir enormt mange muligheter svært fort. For et område med areal 16 hvor det skal plasseres 5 transistorer blir det potensielt over 300 millioner muligheter å undersøke.

Dette forteller oss to ting:

1. Vi kan ikke regne med å løse instanser over en viss, svært begrenset, størrelse.
2. Det er svært ønskelig å finne måter å organisere søket på slik at vi må undersøke så få instanser som mulig.

Backtracking

For å adressere punkt 2 i forrige avsnitt, benyttes "backtracking" (det skulle bli *tilbaketrekking* eller *tilbakesporing* på norsk, men jeg bruker den innarbeidede engelske betegnelsen i fortsettelsen). Dette er en metode for effektiv søking, og er beskrevet blant annet i (Golomb & Baumert, 1965).

Backtracking er en teknikk som benyttes til å søke i store — potensielt uendelige — datastrukturer. Isteden for å generere alle mulige løsninger, for så å finne ut hvilke som oppfyller kravene, genererer man et tre hvor løsningene ligger i bladnodene. Ved å bruke backtracking kan man nå komme tilbake til en tidligere posisjon i søketreet, for f.eks. å prøve en annen løsning.

Oppgaver som skal løses av backtracking må ha følgende generelle form:

> Vi har et produktrom $X_1 \times X_2 \times \ldots \times X_n$ av n utvalgsrom $X_1, X_2, \ldots, X_n$ (som kanskje er det samme rommet).
>
> Vi ønsker å finne en løsning $(x_1, x_2, \ldots, x_n)$ som maksimerer/minimerer en kostnadsfunksjon $\phi(x_1, x_2, \ldots, x_n)$.
>
> Det kan godt være slik at kostnadsfunksjon bare returner 0 eller 1, essensielt "ja" eller "nei". Slik er blir den f.eks. brukt av meg.

Backtracking fungerer ved å bygge opp løsninger en komponent av gangen og bruke en modifisert kostnadsfunksjon for å se om denne løsningen har en mulighet for å bli optimal.

Hvis $(x_1, x_2, -, -, \ldots, -)$ er nødt til å bli suboptimal, kan man hoppe et del mulige test vektorer uten å se over nærmere på dem. Spesifikt kan man hoppe over

$$\prod_{i=3}^{n} M_i = \frac{M}{M_1 M_2} \text{test-vektorer}$$

Et problem med backtracking er at det er vanskelig å analysere hvor lang tid en backtrackingsalgoritme vil bruke. Det vil si, som oftest er det greit å måle ytelsen i O-notasjon, fordi vi da skal gi en formel for hvor lang tid algoritmen skal ta i verste tilfelle. Vanligvis vil dette være noe eksponentielt. Problemet er at dette ikke gir noe riktig bilde av den gjennomsnittlige kjøretiden, som ofte vil kunne være relativt lav. En analyse av gjennomsnittlig kjøretid er svært vanskelig å utføre, og ganske små endringer i algoritmen vil kunne føre til vidt forskjellige resultater. En mulig løsning på dette, er en metode som er beskrevet i (Knuth, 1975) som gir en probabilistisk algoritme for å estimere tidsforbruket. Den baserer seg på å gå tilfeldig nedover en gren av treet, og se hvor mange noder vi støter på underveis. Dette er en ganske enkel metode, som ved relativt få gjentagelser vil fange inn det samme som en gjennomsnittlig kjøring.

2.6.3 Detaljert ruting i minimoduler

Etter at vi har funnet plasseringen til transistorene innenfor en minimodul, må vi se om det lar seg rute. Vi ønsker egentlig å rute nøyaktig, selv om det er tidkrevende. En artikkel som har backtracking-algoritme som kan settes opp til å rute nøyaktig, er (Agrawal & Brauer, 1977).

2.6.4 Lee-algoritmen

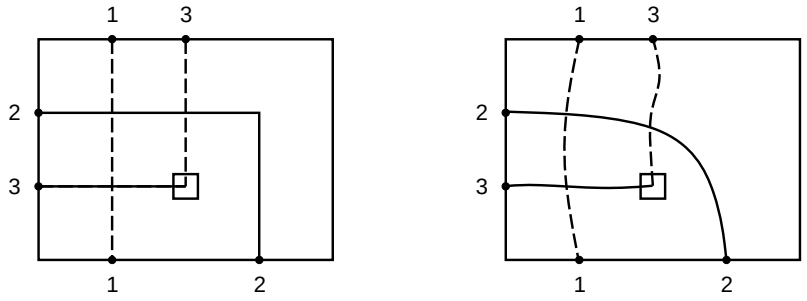
Denne algoritmen er en såkalt labyrintruter (eng. *mazerouter*). Den finner korteste vei mellom to punkter, eller med en modifikasjon, mellom to mengder av punkter. Algoritmen har orden $O(n^2)$, og bruker mye minne. Den ble foreslått i (Moore, 1959) og tilsvarende i (Lee, 1961). Dette er ikke en komplett ruter, men en algoritme som kan brukes som en del av en ruter. En komplett ruter må kunne rute flere nett som konkurrerer om det samme arealet.

2.6.5 Topologisk ruting

Mange rutingalgoritmer lager relativt mange viaer. Som nevnt i avsnitt 1.2.6 er dette uheldig siden viaer fører til dårligere ytelse og øker sjansen for feil ved produksjonen, som nevnt i avsnitt 1.2.6.

Derfor har det blitt laget flere programmer for å forbedre en eksisterende rutingløsning ved å fjerne viaer. Disse programmene er selvsagt nyttige, men den endelige løsningen vil naturlig nok være ganske avhengig av kvaliteten på løsningen man startet med.

(Hsu, 1984) tar opp dette problemet, og beskriver en algoritme som foretar rutingen og via-minimaliseringen samtidig. I artikkelen brukes figur 2.26 for å forklare ideen bak metoden.



(a) Den konkrete løsningen.

(b) Den topologiske informasjonen fra løsningen.

Figur 2.26: Et rutingproblem med tilhørende løsning.

Den viaen som er lagt inn i figuren kan ikke fjernes uansett hvor stor plass man har til rådighet, pga måten nettene krysser hverandre på. Ved å se på denne typen topologiske informasjon vil man kunne finne en nedre grense for antall viaer som trengs. Denne innsikten motiverer derfor til å splitte problemet opp i to faser:

- Topologisk ruting

  Finn en topologisk løsning med minimum antall viaer.

- Geometrisk avbildning

  Projiser den topologiske løsningen ned i et rektilineært plan og bruk så lite område som mulig.

Det spesifikke problemet som blir behandlet i artikkelen tillater bare terminaler langs kantene. Dette forenkler problemet, fordi topologisk sett er det da bare rekkefølgen av terminalene som skiller to forskjellige instanser fra hverandre. Derfor er ikke algoritmen direkte overførbar til problemer der man kan ha terminaler i rutingarealet.

Jeg skal ikke gå inn på detaljene i arbeidet, men resultatet er en algoritme som minimaliserer antall viaer fra et globalt synspunkt. Den problemformuleringen som blir brukt tillater bare terminaler på kantene av rutingarealet, ikke i midten.

Algoritmen gjør ingen forsøk på å minimalisere antall viaer på hver enkelt leder. Dette kan også være ønskelig, spesielt for lange/kritiske nett.

2.6.6 Mighty — en koblingsboksruter

Mighty er en videreutvikling av YACR II som igjen er utviklet fra YACR (Yet Another Channel Router). Mighty er en koblingsboksruter som bruker "rip-up and reroute" teknikker for å sikre god kvalitet på løsningene. Den er fleksibel med hensyn på formen til rutingarealet, og ved at terminaler ikke er begrenset til å ligge langs kantene. Opplysningene i dette avsnittet er stort sett hentet fra (Shin & Sangiovanni-Vincentelli, 1986, 1987), og i mindre grad fra kildekoden. Algoritmen som brukes består av fire hoveddeler:

Veifinner Søker etter veier med lav kostnad.

Veiplasserer Plasserer veier, som veifinneren har foreslått, ut på rutingområdet.

Svak modifikasjon Gjør små endringer for å forbedre forbindelsene.

Sterk modifikasjon Fjerner subnett slik at blokkerte nett kan fullføres.

Veifinner. I denne fasen blir hvert nett prosessert i den rekkefølgen de kommer i innfilen. For hver terminal blir det startet en labyrintruter (eng. *maze router*) for å finne den veien som har lavest kostnad og som samtidig forbinder to terminaler. Veiene blir lagt i en liste sortert på økende kostnad.

Kostnaden måler hovedsaklig veilengde, men tar også hensyn til andre momenter. F.eks. blir det å bytte lag straffet med ekstra kostnad, for å minimalisere antall via-kontakter. Tilsvarende er det billig å utvide en vei i horisontal retning i det ene laget, mens det er billig å utvide i vertikal retning i det andre laget.

Veiplasserer. Etter at alle nettene er prosessert av veifinneren, tar veiplassereren over. Den kikker på veien som ligger først i listen over veier, og plasserer den ut dersom det er mulig, dvs. at det ikke noen overlapp med andre veier som allerede er lagt ut. Hvis det er overlapp blir veifinneren kalt på igjen, for å finne en vei som forbinder to av subnettene i det aktuelle nettet. Nå blir det som allerede er plassert ut tatt hensyn til. Hvis det blir funnet en vei, blir den bare plassert ut dersom kostnaden er innenfor en gitt grense fra den optimale. "Optimal" vil her si den kostnaden veien hadde fått dersom det ikke hadde vært plassert ut andre veier tidligere.

Svak modifikasjon. Hvis det ikke ble funnet noen vei med rimelig kostnad vil den svake modifikasjonsfasen begynne. Den vil dytte rundt på andre nett for å lage plass til en godtakbar vei for problem-nettet. Hvis dette ikke er nok, vil den sterke modifikasjonsfasen bli satt igang.

Sterk modifikasjon. Nå vil utplasserte nett bli fjernet igjen, for å fullføre de blokkerte nettene. Her er det gjort endel for å unngå oscillasjon som lett kan skje hvis man fjerner plasserte nett ukritisk. Det er denne fasen som har gjort at Mighty har fått betegnelsen "A rip-up and reroute detailed router".

Vurdering

Den gjør det (ifølge forfatterne) svært bra i forhold til andre kjente algoritmer. Stort sett bruker Mighty en god algoritme for å løse den oppgaven den er satt til. Spesielt er det *kombinasjonen* av svak og sterk modifikasjonsfase som er nytt. En teknikk man kanskje kunne ønske seg at Mighty hadde benyttet, var at den hadde hatt forsinket lagtilordning som beskrevet i avsnitt 2.6.7.

Et problem med Mighty som ikke har så mye med algoritmen å gjøre som med implementasjonen, er at den bare opererer med to lag til ruting. CETUS (Sun, 1989) bruker Mighty som ruter, og kom dermed opp problemer med denne begrensningen. Forfatteren endret derfor Mighty til å rute i tre lag istedenfor bare to.

2.6.7 BEAVER

BEAVER er en koblingsboksruter (Cohoon & Heck, 1988) som oppnår gode resultater, og som minimaliserer bruken av viaer og lengden på lederne.

BEAVER består av fire deler som blir kjørt etter tur: en hjørneruter, en linjesveip-ruter, en trådruter og en lagtilordner.

Hvis en forbindelse først har blitt lagt ut, vil BEAVER aldri rerute den. Til gjengjeld undersøker den flere alternativer før den tilordner et nett til en bestemt posisjon.

Hjørneruteren finner forbindelser mellom par av terminaler på tilstøtende sidekanter. Eventuelle forbindelser som blir laget vil bare skifte retning en gang.

Linjesveip-ruteren bruker metoder fra "computational geometry" til å finne forbindelser mellom to subnett. Underveis vil hjørneruteren bli startet for å forbinde eventuelle nye hjørneforbindelser som har blitt mulige å løse.

Trådruteren er en labyrintruter (av typen nevnt i avsnitt 2.6.4) som ikke har noen begrensninger på hvilke forbindelser den skal lage, og som derfor vil finne en forbindelse dersom den finnes.

Under hele prosessen blir bestemmelsen av hvilke lag lederne skal bruke, forsinket så lenge som mulig. Det er bare når to forbindelser krysser hverandre at lederne får tilordnet lag. Dette gjør at flere muligheter holdes åpne lenger, og det fører til bedre løsninger.

Til slutt må alle lederne få tilordnet lag, og det blir gjort av lagtilordneren på en slik måte at antall viaer blir minimalisert.

Kapittel 3

Beskrivelse av eget arbeide

I dette kapitlet vil jeg ta for meg det jeg selv har gjort. Jeg vil beskrive implementasjonen av de to programmene jeg har skrevet — programmet Cell, som foretar plassering av små instanser og et program for k-veis partisjonering. Jeg vil også se på en del forhold som man må ta hensyn til når et verktøy av denne typen skal utvikles, også hvilke valg jeg utfra disse punktene ble motivert til å gjøre. Som en del av dette har jeg valgt å se grundig på hvordan det å stille spørsmålet på riktig måte påvirker løsningen av et problem.

3.1 Målet med oppgaven

Målet med oppgaven er å bidra til å løse noen av problemene i forbindelse med automatisk utlegg av transistorer. Oppgaven bygger i stor grad på arbeidet til Kjell Øystein Arisland (Arisland, 1989). Det nye i hans arbeid, er forsøket på å finne en organisering av utleggsproblemet som kan gi opphav til bedre resultater enn tidligere metoder. Andre har brukt "splitt og hersk" prinsippet tidligere, men her er noe av poenget at det nederste nivået løses optimalt.

3.2 Arbeidet med oppgaven

I det jeg startet med oppgaven var det meningen at flere skulle arbeide med dette verktøyet, og tilstøtende problemer. Jeg skulle i hovedsak se på algoritmer til bruk for partisjoneringsfasen, mens andre skulle ta seg av resten. Av forskjellige grunner ble det aldri flere som arbeidet med dette. Etter en stund ble jeg derfor også trukket inn arbeidet med å lage et program for å legge ut minimodulene optimalt.

Jeg startet med å se på Arislands program for å lage legge ut minimoduler, på grunnlag av små nettlister. Det var midt mellom to omskrivninger. I en tidligere versjon hadde plasseringen fungert, men slik programmet var når jeg overtok det, var det lagt til noe kode for ruting, men programmet kompilerte ikke.

Etter å ha satt meg inn i hans program, startet jeg arbeidet med å skrive et nytt program, bygget på de erfaringene som nå var gjort. Ved å skrive programmet i mer objektorientert stil, og med C++ som implementasjonspråk, håpet jeg å unngå de problemene som hadde ført til at det forrige programmet hadde vist seg vanskelig å vedlikeholde, og vanskelig å utvide.

Dette var en stor oppgave, av flere grunner.

- Jeg måtte sette meg inn i et nytt programmeringspråk, C++.

- For å visualisere resultatene, måtte jeg lære meg et grensesnitt mot vindussystemet X11. Jeg valgte InterViews til dette formålet som beskrevet i avsnitt 3.5.3.

- Jeg la mye arbeid ned i å gi programmet en god struktur. Dette innebærer at mye av koden skal kunne anvendes også til andre oppgaver. Spesielt var det aktuelt å se om rutingen kunne bruke de samme mekanismene, men jeg implementerte også dronningoppgaven med det samme systemet for å demonstrere at det var generelt (beskrevet i avsnitt 3.5.5).

- For at rutingen skulle bli rask nok, var det viktig å gjøre avskjæringer så tidlig som mulig. Det er vanskelig.

Jeg begynte også å arbeide med å rute minimodulene i samme stil som plasseringen. Jeg startet med de ideene Arisland hadde, nemlig å gjøre ruting i samme stil som plasseringen. Det vil si å lage en algoritme som løser problemet optimalt, vha. backtracking.

Problemet med en slik fremgangsmåte, er å få det effektivt nok. Derfor forsøkte jeg å finne en måte å avskjære suboptimale løsninger så tidlig som mulig, uten at man går glipp av de optimale løsningene. Det å lage et program med mange slike optimaliseringer er tidkrevende, så for å få noe på lufta, valgte jeg å se på eksisterende rutere, for å se om jeg kunne koble meg opp mot dem.

Til slutt har jeg returnert til det opprinnelige målet for oppgaven, ved å se på kriterier og algoritmer for partisjonering. Dette er et stort felt, og jeg har gått grundig inn i litteraturen på området. Ut fra det som har vært gjort tidligere, identifiserer jeg flere svakheter med tidligere kriterier. Ved å integrere fasene på en helt ny måte, kommer jeg frem til et kriterium som er mer i samsvar med det vi egentlig ønsker å utføre. Dette gir grunnlag for bedre løsninger. Jeg har også implementert en algoritme som gjør dette.

I tillegg til alt dette, har jeg kommet med noen forslag til videre arbeid på de resterende delende av verktøyet. Dette gjelder først og fremst rammeverket som kobler partisjoneringen sammen med Cell-programmet.

3.3 Valg i forbindelse med arkitekturen

3.3.1 Automatisk kontra interaktivt

Når det gjelder valget mellom å gjøre et verktøy automatisk og interaktivt, er det noen poenger jeg vil anmerke. Dette er en glidende skala, fra helt automatisk, til svært interaktivt — som f.eks en utleggseditor.

Et skille går mellom de verktøyene som er mer å regne som avanserte manuelle utleggsverktøy, og de som er automatiske verktøy som faller tilbake på brukeren når noe går galt. Dette skillet vil også avspeile seg i den omgivelsen de arbeider i, og i den tiden de kan tillate seg å bruke. En interaktiv bruker trenger rask respons for å arbeide effektivt. Hvis en operasjon tar mer enn noen sekunder, blir den interaktive prosessen lite tilfredstillende for brukeren. Et automatisk verktøy kan i større grad tillate seg å bruke lenger tid. Hvis man faktisk klarer å lage et utlegg som er bedre enn det en menneskelig designer klarer, er det f.eks ikke *så* farlig at en kjøring går natten over.

Hvis det automatiske systemet fungerer bra, er det liten grunn til å involvere en bruker. Mange verktøy forsøker å løse et problem automatisk, men hvis det blir for vanskelig vil de falle tilbake på brukeren. Dette forutsetter at brukeren er en ekspert.

I noen tilfeller vil dette være OK. Hvis man har svært strenge grenser for arealet som er tilgjengelig, kan det være greit å kreve at en ekspert gjør resten. Dette gjelder f.eks (Domic et al., 1989).

I andre tilfeller er det mer naturlig å la alt bli gjort automatisk, og så heller forbedre det etterpå. I denne oppgaven er det uansett forhåndsbestemt at det er automatiske algoritmer som er temaet.

3.3.2 Parallellisering

Som nevnt i avsnitt 2.3.3 er det foreslått flere partisjoneringsalgoritmer som egner seg for parallellisering. Dette kommer av at sekvensielle maskiner aldri er så raske som vi ønsker. Ved å sette flere maskiner sammen (eller ved å sette flere CPUer inn i en maskin) kan man fordele arbeidet, og få resultater fortere. Det vi ønsker er at hvis n maskiner arbeider med problemet, vil det ta $1/n$-del av tiden en maskin ville ha brukt. Det er slett ikke alle algoritmer som lar seg parallellisere så godt. Hvis en algoritme er sekvensiell av natur, eller krever mye kommunikasjon mellom forskjellige deler av algoritmen, er det ikke sikkert at man får noen vesentlig forbedring ved parallelliseringen.

Det er flere måter å parallellisere på, avhengig av hva slags maskin man kjører på, og avhengig av algoritmen man bruker. Noen algoritmer som krever mye kommunikasjon mellom delene kan allikevel få forbedret ytelse ved å benytte maskinarkitekturer optimalisert for parallellisering. Men den enkleste måten å parallellisere på, er hvis problemet lar seg dele opp i flere uavhengige oppgaver som er omtrent like beregningskrevende. Da kan man la hver prosessor/maskin arbeide på sin uavhengige oppgave. Dette er et enkelt konsept, og egner seg hvis kommunikasjon er relativt dyrt, som f.eks hvis man har tenkt å bruke vanlige maskiner forbundet med lokalnett.

Jeg skal her undersøke hvor egnet strukturen i denne oppgaven er til parallellisering. Strukturen er som følger:

- Partisjonering
- Gå fra små nettlister til minimoduler
- Sett sammen minimodulene (ruting, plassering av brønnkontakter, osv.)

For partisjoneringsfasen er det foreslått flere parallellalgoritmer. De fleste av disse er basert på bipartisjonering, men algoritmen som er beskrevet i (Shin & Kim, 1993) skulle kunne modifiseres k-partisjonering.

Av disse fasene, er det oppgaven med å gå fra små nettlister til minimoduler som er den mest omfattende. Den fasen er svært egnet til en slik grovkornet parallellisering som er nevnt over. Det er mange oppgaver (hvis vi antar 200 transistorer fordelt på fem transistorer i hver del, vil gi oss 40 deler), som alle er uavhengige, eller som kan gjøres uavhengige uten at kvaliteten på løsningene går vesentlig ned. Grunnen til at de kan gå litt ned, er at hvis man legger ut minimodulene etter tur, kan utlegget av de senere minimodulene benytte kunnskapen om hvordan de tidligere minimodulene har blitt lagt ut. Det viktigste vil være å få fastslått plasseringen av terminalene som grenser inn til modulen som man er i ferd med å legge ut.

Den siste fasen med å sette sammen minimodulene, består av forskjellige oppgaver. Hvor lett det er å parallellisere disse oppgavene, varierer litt. Globalrutingen kan nok være litt problematisk å parallellisere godt, mens lokalrutingen blir tilsvarende utlegget av minimoduler.

Jeg skal ikke følge opp dette videre i denne oppgaven, men det er nyttig å vite at systemet med liten innsats (ved å kjøre utlegget av minimoduler på forskjellige maskiner/prosessorer) ville kunne bli relativt godt egnet til parallellisering. Med litt større innsats (ved også å bruke parallellalgoritmer til partisjoneringen og mer av rutingen) vil svært store deler av verktøyets funksjoner få glede av parallellkjøring.

3.3.3 Forholdet mellom høyde og bredde

Forholdet mellom høyde og bredde kan være viktig hvis det bare er plass til en lang og tynn modul på en gitt plass. Forholdet mellom høyde og bredde kalles aspektforholdet (eng. *aspect ratio*). Måten programmet håndterer ønsket om et bestemt aspektforhold er å lage et rektangel satt sammen av kvadratiske småruter. Ved å sette sammen rektangelet av f.eks 4 ganger 9 småmoduler, vil også det totale rektangelet få omtrent dette forholdet mellom høyde og bredde. Det vil ikke bli nøyaktig, av følgende årsaker.

- Småmodulene ikke er helt kvadratiske.

  De vil ha høyde/bredde forhold hvor den ene siden kan ha en enhet mer eller mindre enn den andre.

- Det vil sannsynligvis bli ekspandert ulikt mellom minimodulene.

  Det virker rimelig å legge mer enn en minimodul i en brønn, slik at man ikke kaster bort plass på områder hvor det ikke kan legges transistorer. Dessuten vil også plasseringen av strømforsyningen påvirke dette.

- Eventuell kompaktering av modulen vil kunne endre aspektforholdet helt til slutt.

En liten feil i forhold til ønsket forhold mellom sidene vil uansett sjelden være noe problem. Siden vi allikevel ikke gir noen garanti for hvor stort areal modulgeneratoren skal bruke, kan vi eventuelt gi riktig aspektforhold ved å bruke litt ekstra plass i den ene retningen.

Basert på fremgangsmåten med å skape riktig aspektforhold ved å sette sammen kvadratiske småruter, kan vi sette opp følgende kriterier for å velge passende forhold mellom høyde/bredde:

- Både høyde og bredde må være heltall.

- Det vil sannsynligvis måtte være en minimumsgrense for hvor få minimoduler det kan være langs en side. Dette kravet har sammenheng med plassering av brønner, og hvorvidt det er et en-til-en forhold mellom brønner og minimoduler. Dette behandles nærmere i neste avsnitt, men i resten av dette avsnittet vil jeg anta at minimumsgrensen tilsier at ingen side kan ha mindre enn to minimoduler.

- Ved å vende modulen $90°$, vil høyde bli bredde og omvendt. Derfor vil jeg, uten tap av generalitet, bare behandle aspektforhold mindre enn 1.

Følgende tabell kan lages ved hjelp av en multiplikasjonstabell, og inneholder antall minimoduler, og hvilke forhold mellom kantene som kan gi en slik størrelse:

Areal	Kan realiseres ved
4	2×2
6	2×3
8	2×4
9	3×3
10	2×5
12	$2 \times 6, 3 \times 4$
14	2×7
15	3×5
16	$2 \times 8, 4 \times 4$
18	$2 \times 9, 3 \times 6$
20	$2 \times 10, 4 \times 5$
22	2×11
24	$2 \times 12, 3 \times 8, 4 \times 6$
25	5×5
26	2×13
27	3×9
28	$2 \times 14, 4 \times 7$

Areal	Kan realiseres ved
30	$2 \times 15, 5 \times 6$
32	$2 \times 16, 4 \times 8$
33	3×11
34	2×17
35	5×7
36	$2 \times 18, 3 \times 12, 4 \times 9, 6 \times 6$
38	2×19
39	3×13
40	$2 \times 20, 4 \times 10, 5 \times 8$
42	$2 \times 21, 3 \times 14, 6 \times 7$
44	$2 \times 22, 4 \times 11$
45	$3 \times 15, 5 \times 9$
46	2×23
48	$2 \times 24, 3 \times 16, 4 \times 12, 6 \times 8$
49	7×7
50	$2 \times 25, 5 \times 10$

Måten dette kan brukes på, er å se hvor mange transistorer vi har totalt, la oss si 200. Så deler vi dette på det antallet transistorer som erfaringsmessig passer best til å la seg løse av programmet som legger ut de små nettlistene, la oss si 5. Dette gir 40 nettlister. Hvis man er bedt om å fremskaffe en modul med et aspektforhold på ca 1 får vi et valg: enten kan vi godta å lage et rektangel med sidekanter 5 og 8 (som gir et forhold på 0.625), eller så kan vi justere størrelsen på noen av de små nettlistene slik at vi får litt flere, eller litt færre deler. I dette tilfellet vil det kanskje være naturlig å lage 42 nettlister, plassert i et 6x7 rektangel, med et forhold på ca. 0.86. Dette valget vil også kunne bli styrt av programmet som skal bruke den endelige modulen, ved at det angir grenser for hvilke aspektforholdet som blir godtatt.

Dette er en enkel måte å gi det kallende programmet kontroll over aspektforholdet på, som ikke øker kompleksiteten så mye og som ikke går utover kvaliteten på de ferdige utleggene. Hvis det er veldig viktig å ha detaljkontroll over aspektforholdet, kan man tilpasse aspektforholdet på minimodulene for å få enda mer nøyaktige resultater. Dette vil imidlertid bli mer komplisert, og må testes grundig for å kontrollere at dette ikke vil gå utover kvaliteten.

3.3.4 Plassering av brønner

Hvis vi ikke tar hensyn til brønner og P- og N-transistorer før helt til slutt, kommer mye plass til å gå med til brønnkanter hvor man ikke kan plassere transistorer. For å unngå dette, må man sørge for at brønnene har en viss størrelse, med flere transistorer av samme type i hver brønn.

For at utlegget skal bli kompakt, må transistorene innenfor en brønn ha flest mulig koblinger som holder seg innenfor brønnen. Derfor virker det rimelig å kombinere dette med partisjoneringsfasen, hvor det nettopp er målet å dele opp i mindre deler slik at transistorene hører sammen. Det betyr at algoritmen som foretar partisjonering/gruppering må sørge for at bare en type transistorer havner i hver del.

Et spørsmål er om det skal være et en-til-en forhold mellom brønner og minimoduler. Dette kan ikke fastslåes endelig, uten å vite hvor store utlegg en minimodul kan håndtere.

Men sannsynligvis vil minimodulene være mindre enn det som er en hensiktsmessig størrelse på brønnene. Det er naturlig å la en brønn bestå av et multippel av minimoduler, f.eks to eller tre. Da slipper ihvertfall delprogrammet som skal legge ut de små nettlistene å ta hensyn til brønner. Det vil gjøre delprogrammet enklere.

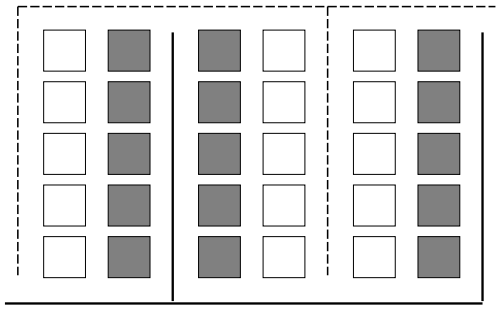
3.3.5 Plassering av spenningsforsyning og jord

Plasseringen av brønner påvirker hvordan minimodulene bør legges ut. En annen faktor som også virker inn på hvordan strukturen på et utlegg skal være, er plasseringen av lederne for spenningsforsyning og jord. Disse skal typisk kobles til svært mange transistorer, og bør derfor spesialbehandles.

En god struktur er den som blir brukt i BBC som blir beskrevet i avsnitt 2.1.4, ved å la V_{dd} og jord alternere mellom radene, slik at følgende mønster vil gå igjen:

V_{dd}, P-transistorer, N-transistorer, Jord

Ved å bruke denne strukturen sammen med det som er nevnt tidligere, vil vi kunne få utlegg av den typen som figur 3.1 demonstrerer.



Figur 3.1: Oppbygningen av utlegg med den stilen som blir brukt i denne oppgaven. De stiplete og heltrukne linjene angir V_{dd}/jord, mens fargen på minimodulene angir hvilken type transistorer de består av, og dermed også hvor brønnene ligger.

Denne figuren viser også hvorfor det ikke er heldig med utlegg som består av f.eks 1×10. Enten blir brønnene for små (hvor små de blir er avhengig av hvor mange transistorer Cell kan håndtere), eller så vil P-transistorene vil ikke komme nær de tilhørende N-transistorene. (At det P- og N-transistorer "hører" sammen, blir gått nærmere inn på i avsnitt 3.7.13.)

3.3.6 Oppdeling i faser

Som nevnt i avsnitt 2.2, er det ønskelig å finne en oppdeling i faser som tillater at vi tar hensyn til det meste av det som er relevant, samtidig som problemet lar seg løse i rimelig tid.

Hvis vi bare slår sammen to faser for å løse dem på en gang, blir det fort for vanskelig. Det man må unngå, er at de to fasene knyttes sammen så tett at det blir en for sterk vekselsvirkning mellom dem. Hvis den første fasen gjør et galt valg, og den andre fasen finner det ut, er det uheldig om den første fasen må gjøre jobben på nytt. Da har man ingen garanti om at ikke den andre fasen vil finne noe galt med det nye valget også, slik at vi får en lang (kanskje uendelig) sekvens med arbeid som må gjøres om igjen.

Jeg tror at en god struktur kan være å plukke ut visse sider av problemet i en fase, som så kan brukes i den neste. Altså at det er litt mer data som kommuniseres mellom fasene.

Hvis man kan la den første fasen kalle på den andre for å løse et delproblem, så kan det taes hensyn til av den første fasen — uten at noe må gjøres om igjen.

En sammenkobling mellom faser som jeg tror vil kunne være nyttig, er en jeg ikke skal gå nærmere inn på i denne oppgaven. Det er å kombinere informasjon fra den logiske syntesen med utleggsfasen. Dette blir gjort av såkalte "silicon compilers", og jeg tror dette har potensiale til å finne gode løsninger. Men jeg tror det er viktig at de bruker noen av de samme ideene som de bak denne oppgaven, nemlig å forsøke å løse problemene nær optimalt.

3.4 Plassering av transistorer — programmet Cell

3.4.1 Beskrivelse av målet for plasseringsprogrammet

Arbeidet med å detaljplassere transistorene er en liten del av det totale utleggsverktøyet, men det er en ganske viktig del. Målet er at utlegget av minimoduler skal løses optimalt, og derfor må plasseringen av transistorene løses optimalt. Samtidig må man foreta avskjæringer så tidlig som mulig, slik at det ikke skal ta for lang tid.

3.4.2 Beskrivelse av arbeidet med plasseringsprogrammet

Jeg startet med å se på et program som Arisland hadde begynt på. Dette var ikke ferdig, og trengte å skrives om. Faktisk var programmet midt i en omskrivingsprosess, slik at det ikke lot seg kompilere.

Etter å ha brukt litt tid på å sette meg inn i den koden, bestemte jeg meg for å starte på nytt, med endel nye ideer, samtidig som jeg tok med meg erfaringene fra det originale programmet.

3.4.3 Oversikt over plasseringsprogrammet

Som nevnt i avsnitt 2.1.1 fjerner en gitter-representasjon av utleggsarealet unødvendige detaljer. Grunnen til det, er at man kan ha en mye enklere datastruktur. Isteden for at transistorene kan plasseres utover på alle mulige punkter på utleggsarealet, blir det bare mulig å plassere transistorene på noen få faste steder.

Selv med denne begrensningen er det mange forskjellige måter å plassere transistorene. Derfor er det viktig å finne en struktur på problemet som lar oss bruke avanserte avskjæringer til å kutte ned antall løsninger vi studerer nærmere. Denne strukturen er backtracking (se avsnitt 2.6.2). Begge disse punktene, som kommer fra arbeidet til Arisland, er sentrale i programmet som jeg har skrevet.

Måten backtracking blir brukt på i dette programmet er ved at transistorene blir plassert ut i rutingområdet på følgende måte:

> Først blir den første transistoren plassert i første posisjon i gitteret. Hvis den passer der, blir neste transistoren plassert i første posisjon. Hvis posisjonen er opptatt, blir den isteden forsøkt plassert i neste posisjon. Slik går det helt til alle transistorene har blitt plassert eller man ikke har funnet noen løsning.

Underveis blir det gjort avskjæringer. Den første og viktigste avskjæringen sørger for at ingen transistorer kan stå i samme posisjon. Alle avskjæringene blir presentert i detalj under.

Kostnaden med mange avskjæringer

For at backtrackingen skal være effektiv, må man ha gode avskjæringer. Det bør ikke plasseres transistorer i posisjoner som fører til at rutingen ikke lar seg gjennomføre. Samtidig er dette kode som vil bli utført svært mange ganger, så en slik evalueringsfunksjon bør ikke ta for lang tid å utføre. Spørsmålet er om hva som er best av:

1. å ha en enkel, rask og dårlig evalueringsfunksjon som kan kjøres mange nivåer nedover, eller

2. å ha en komplisert, treg og god evalueringsfunksjon som bare kan kjøres på få nivåer.

For å gi spørsmålet det riktige perspektivet, kan jeg nevne at begge måtene vil kunne finne det optimale svaret: Den enkle evalueringsfunksjon vil finne det optimale svaret hvis man går helt til bunns i rekursjonen. Og hvis evalueringsfunksjonen av type 2 er optimal, trenger man ikke å søke mer enn et nivå ned for å finne det optimale svaret. For sjakkprogrammer viser det seg at de som gjør det best for tiden bruker en relativt rask og enkel evalueringsfunksjon av type 1.

Hva som er best for det problemet jeg skal løse er vanskelig å finne ut annet ved prøving og feiling. På grunn av backtrackingen kan små forskjeller i koden gi store forskjeller i kjøretiden.

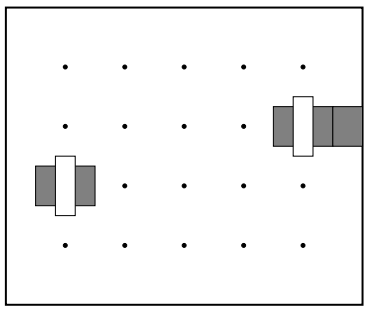
Konkrete avskjæringer

Her skal jeg si litt om de avskjæringene som blir brukt i dette programmet. Plasseringen av transistorene har mye med rutingen å gjøre. En god plassering er slik at utlegget lar seg rute, samtidig som det er lite plass til overs som ikke er i bruk. Derfor er det nyttig hvis plasseringsprogrammet kan estimere om rutingen vil la seg gjøre eller ikke, siden man da slipper å utføre ruting som ikke vil gå opp.

Kun en transistor i hver posisjon Den første avskjæringen er som nevnt ikke direkte relatert til ruting, men isteden til det opplagte faktum at to transistorer ikke kan stå i samme posisjon.

Diffusjon utenfor rutingarealet Som en konvensjon, vil det ikke bli lagt diffusjon utenfor området hvis det ikke lå der fra før av. Dette vil fjerne noen muligheter, men vil samtidig sørge for at forbindelser til omverdenen normalt ikke vil gå i diff-laget.

Denne avskjæringen sjekker om det kommer til å havne diffusjon utenfor rutingarealet, og vil bare tillate det dersom det allerede er plassert diffusjon fra riktig nett der på forhånd og nettet bare har to terminaler. Disse poengene er illustrert i figur 3.2.

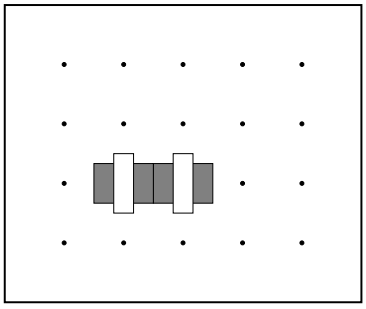


Figur 3.2: Det å plassere den venstre transistoren, som vist her, vil føre til et utlegg som ikke kan rutes. Om transistoren på høyresiden kan plasseres der den står vil være avhengig av om source/drain på transistoren skal tilkobles samme 2-terminals nett som diffusjonen som lå der på forhånd.

Slik programmet blir startet, vil det aldri være diffusjon utenfor rutingarealet, så denne avskjæringen vil i praksis innskrenke rutingarealet: Transistorene langs kantene kan bare ha diffusjonen parallelt med sidekanten, og det vil aldri kunne stå noen transistor i hjørnene.

Kollisjoner med nabopunktene Denne avskjæringen ligner litt på den første, i den forstand at den forsøker å unngå kollisjoner. Siden vi vet at det ikke står noe annet i samme punkt som vi ønsker å plassere i, sjekker vi nå de 8 nabopunktene. Avhengig av hva som ligger der på forhånd vil det kunne være ulovlig å plassere noe nytt så tett inntil. Om det er lov eller ikke, blir avgjort ved å se på informasjonen fra teknologireglene. Slik det er nå, blir de kompilert inn i programmet som verdier i et array, men det vil ikke være noe problem å lese disse verdiene inn ved oppstart. Poenget her er ihvertfall at man kan lage nye slike teknologiregler, som beskrevet i neste avsnitt. Dette avslutter den første delen av avskjæringene, som har direkte med lovligheten av plasseringen å gjøre.

Forbindelser til nabopunktene Den neste avskjæringen går litt nærmere inn på om rutingen vil kunne bli vellykket utifra plasseringen som er gjort. Et eksempel er to transistorer som plasseres med diffusjon inntil hverandre (som i figur 3.3).



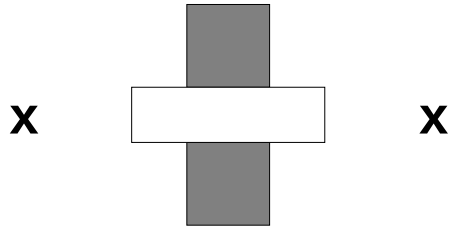
Figur 3.3: To transistorer plassert tett inntil hverandre.

Med de reglene som brukes av programmet vil transistorene være i direkte kontakt. Dette er greit dersom de to transistorene skal kobles sammen, og hvis nettet bare består av disse to terminalene. (Da er det faktisk svært ønskelig, siden det sannsynligvis fører til et kompakt utlegg.) Men hvis disse forutsetningene ikke holder vil plasseringen ikke ha noen sjanse til å føre til et fungerende utlegg.

Merk at denne regelen ikke ligger innebygget i programkoden, men kommer fra teknologireglene. Man kan muligens tenke seg et annet sett med teknologiregler som sier at man kan sette små transistorer så tett og allikevel unngå kontakt, mens store transistorer vil være i kontakt. De reglene som brukes nå skiller ikke mellom store og små transistorer, men det er en mulighet andre teknologiregler kan gi.

Denne avskjæring vil altså ikke sjekke spesifikt om source og drain er sammenkoblet, men vil isteden konsultere teknologireglene, som vil gi det riktige svaret.

Frihetsgrader En avskjæring som Arisland begynte å arbeide med, og som jeg også har sett på, er basert på frihetsgrader. Dette er egentlig et begrep fra spillet Go, og forteller i denne sammenhengen hvor nær en terminal (enten en I/O-tilkobling, eller source, drain eller gate fra en transistor) er fra å bli helt omringet. Et eksempel på hvor dette kan brukes, er en gate-terminal. Gate-terminalen kan tilkobles resten av nettet den tilhører på begge sidene av en transistor, så det vil ikke være ulovlig å sperre for en av sidene. Men hvis man sperrer for begge sider, vil gate-terminalen aldri kunne kobles til nettet sitt, og utlegget vil aldri kunne rutes (som i figur 3.4).

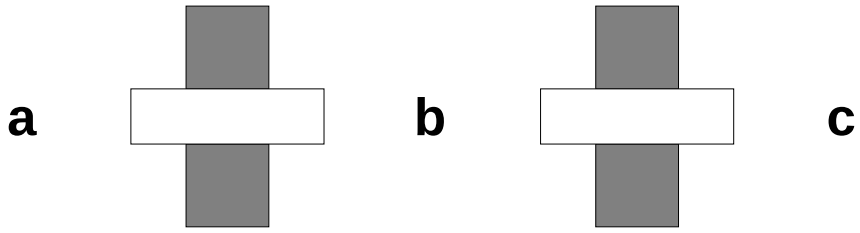


Figur 3.4: Hvis det blir sperret på begge sider av en gate, vil ikke utlegget kunne rutes.

For å fange opp slike situasjoner, kan man holde rede på hvor langt man er fra å sperre terminalene inne, og ikke tillate plasseringer som fjerner den siste frihetsgraden. Dette ligner litt på arbeidet som er gjort med X-rutere, som er illustrert i figur 2.24 på side 48.

Men situasjonen er i grunnen enklere for rutere, fordi i vårt tilfelle blir det stadig plassert nye elementer. Hvis et av de nye elementene faktisk kan tilkobles, vil det være lov å plassere det, selv om posisjonen vil være sperret for andre elementer.

Dette er forsåvidt greit i utgangspunktet. Et spørsmål er om slike frihetspunkter kan deles. Med det mener jeg om et punkt i matrisen kan være frihetspunkt for flere terminaler på en gang, slik det er i figur 3.5.



Figur 3.5: Kan b fungere som et frihetspunkt for begge transistorene?

Hvis man velger en avansert løsning her, kan avskjæringen bli litt sterkere, men samtidig vil den ta lenger tid. Denne avskjæringen har ikke blitt ferdig implementert i programmet Cell.

3.4.4 Teknologi-uavhengighet

Som nevnt i avsnitt 1.2.10 må man alltid forholde seg til et sett med designregler for å lage et utlegg som skal produseres. Men det er ikke ønskelig å binde seg til et bestemt sett med regler. Derfor er det et mål å parametrisere designreglene så mye som mulig, slik at disse kan leses

inn ved oppstart. På den måten vil programmet kunne bruke et nytt sett med designregler — sålenge det ikke er altfor forskjellig fra den grunnleggende modellen — ved at man skriver designreglene på formatet som programmet bruker.

Ekspertsystem

En fordel av å parameterisere designreglene er at dette programmet kan brukes for flere teknologier, også morgendagens. Men i tillegg kan de brukes til å finjustere oppførselen til programmet.

Det er et skille mellom *designregler*, som er regler som hører til en bestemt fabrikk eller produksjonslinje, og *teknologiregler*, som er reglene som blir lest av programmet Cell. Teknologireglene må avspeile alle kravene i designreglene, men kan også inneholde regler som bare gir mening for Cell. En person som kjenner designreglene godt, kan formulere noen teknologiregler som f.eks tillater flere forskjellige transistor-størrelser.

Jeg skal overlate definisjonen av nye teknologiregler til ekspertene, men jeg har lagt inn stor fleksibilitet i Cell nettopp med dette for øyet.

På denne måten får man ideelt et slags manuelt ekspertsystem. Siden det kan være tungvint å spesifisere slike mer innfløkte regler direkte inn i en datafil med en vanlig tekstbehandler, kan det være ønskelig å lage et grafisk program som gjør at man kan spesifisere parametrene så visuelt som mulig.

Teknologimodul

Det kan være hensiktsmessig å la alt som har med designreglene å gjøre ligge i en egen modul. De andre delene av programmet kan da forholde seg til denne modulen isteden for til skiftende designregler. Dette er imidlertid ikke lett, ihvertfall ikke hvis programmet også skal bli effektivt. Man kan ikke optimalisere koden med hensyn på en bestemt teknologi, slik det ellers kunne gjøres.

Slik det er nå, blir noen slike regler kodet inn i programmet som verdier i en matrise, men det er trivielt å la disse reglene bli lest inn ved oppstart isteden.

Varierende antall metall-lag er ikke implementert, men det er mest for å avgrense problemet. Det skulle ikke være noe prinsipielt problem å legge inn støtte for det.

3.4.5 Styring av Cell

Cell blir kalt med et forhåndsbestemt areal, og skal bare se om det er stort nok. Arbeidet med å finne det minste arealet som tillater at den lille nettlisten lar seg legge ut, blir lagt på rutinen som kaller på Cell. Den vil typisk kalle på Cell med et areal som vil være i minste laget. Hvis det går bra, er det greit, hvis ikke vil Cell bli startet på nytt med et areal som er litt større. Det første arealet som Cell klarer å lage et utlegg med, vil derfor være det minste som går.

Vi kan også tenke oss at vi bruker en variant av binærsøk, for å finne det minste arealet som går bra. Binærsøk er normalt svært effektivt, men i dette tilfellet er det ikke sikkert at det fungerer like bra. Det er fordi det sannsynligvis vil ta kortere tid å finne ut at det ikke lar seg legge ut, enn det tar å finne en løsning dersom den finnes. Det er fordi et lite areal vil føre til mange avskjæringer, mens et stort areal krever mer arbeid før man vet noe sikkert.

Men uansett hva vi gjør, er det viktig å ha gode estimater for hva som blir det minste arealet. Det kan spare mange unødvendige kjøringer, og dermed mye tid. Slike estimater kan vi skaffe ved å kjøre Cell på mange forskjellige instanser, og se hvor store areal som trengs til

hvor mange transistorer, og hvordan oppbygningen av nettlisten vil påvirke dette. Hvis to av transistorene kan settes tett inntil hverandre, vil nettlisten sannsynligvis trenge mindre areal enn ellers.

3.5 Implementasjonen av Cell

I dette avsnittet skal jeg skrive litt om det praktiske arbeidet med programmet, og deriblant hvilke valg av språk og programeringsomgivelse som ble gjort.

3.5.1 Valg av språk

Tradisjonelt har C vært valgt til slike prosjekter tidligere. Til dette prosjektet valgte vi C++. Her skal jeg se litt på bakgrunnen for dette valget.

Grunner til å velge C++

Ligner C. Språket kan brukes som en forbedret utgave av C (med f.eks. inline-funksjoner og sterkere typing) selv om man ikke er interessert i å benytte seg av klasser. En C++-kompilator kan kompilere de fleste C programmer uten endringer. Dette vil ikke være noe stort poeng i lengden, men likheten med C er en viktig grunn til at mange begynner å bruke C++.

Ligner SIMULA. Dette er en spesielt stor fordel her ved instituttet siden alle lærer SIMULA i grunnkursene. Men siden SIMULA har mange gode egenskaper er dette et pluss uansett. Fordelene med SIMULA er at det støtter objekt-orientering (det har C++ arvet fra SIMULA) og har innebygget søppeltømming (eng. *garbage collection*) av minnet (det har C++ ikke tatt med seg). De viktigste ulempene med SIMULA er den lave utbredelsen i verden forøvrig, hastigheten på den ferdige koden og at det finnes få ferdige biblioteker, f.eks. for grafikk mot vindusystemet X11. Her har C++ klare fordeler (bl.a. med InterViews som er beskrevet i avsnitt 3.5.3).

Objekt-orientering. Dette skal jeg si mer om i avsnitt 3.5.2.

Den ferdige koden eksekverer raskt. C er regnet som et "raskt" språk, og siden C++ er et overbygg på C, sier det seg selv at det er mulig å skrive rask C++-kode. Men hvis man bruker de mer avanserte mulighetene i C++ (som f.eks. virtuelle funksjoner) vil ting begynne å gå tregere.

Industristandard. Mange skjønner koden du skriver, og mange kan hjelpe om du blir sittende fast. Mange utvikler nye ting til, og i, C++.

Passer inn i resten av arbeidsmiljøet. Her tenker jeg på ting som at `make` fungerer greit, og at editoren Emacs har støtte for C++, o.l. Hvis man hadde blitt tvunget til å bruke en egen omgivelse (slik man i praksis blir i f.eks. Smalltalk) ville det ikke vært så aktuelt.

Grunner til *ikke* å velge C++

Det er flere grunner til ikke å velge C++ også. Noen av dem kommer her:

Ligner C. C kan være ganske kryptisk, og det er lettere å skrive komplett uforståelige ting i C (og dermed også C++) enn i andre språk. Det er faktisk en årviss konkuranse om hvem som kan skrive de mest uforståelige programmene kalt "The International Obfuscated C Code Contest". Her kommer et eksempel som løser dronningproblemet (se avsnitt 3.5.5 for en beskrivelse hvis dette problemet er ukjent):

```
v,i,j,k,l,s,a[99];
main()
{for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<s,j+=(v=j<s&&(!k&&!!printf
  (2+"\n\n%c"-(!l<<!j)," #Q"[lv?(lj)&1:2])&& ++l||a[i]<s&&v&&n%v-
  i+j&&v+i-j))&&!(l%=s),v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);}
```

Selv om dette er et program som er skrevet for å være uforståelig, kom inspirasjonen fra et ekte program (Unix kommandoen `finger`), og uansett vil det være vanskelig å skrive noe som er *så* kryptisk i SIMULA eller andre språk.

Mangler søppeltømming av minnet. Det at man må ta seg av dette selv er en stor kilde til feil. Riktignok har programmer som håndterer deallokering eksplisitt mulighet for å være litt mer effektive enn ved bruk av søppeltømming, men programmerere gjør erfaringsmessig mange feil akkurat på dette området. Dessuten har det skjedd store forbedringer på ytelsen på søppeltømmere den siste tiden.

Språket endrer seg. Dette gjør at det er forskjellige versjoner av kompilatorer og biblioteker. Det blir mer å ta hensyn til. Koden blir fort "gammel", og må skrives om jevnlig hvis man alltid skal ha et program som følger dagens standard.

Det finnes få gode debuggere. Dette vil nok rette seg innen kort tid, men det hjelper lite når koden skal skrives nå.

C++ er stort og uoversiktlig, og har mange unntak. Det er fremdeles et stykke igjen til virkelig store språk (som f.eks. Ada), men det er endel unntak som gjør at språket ikke er så "lite og rent" som C.

Resultat av språkvalg

Det endte med at C++ ble valgt, fordi min vurdering var at fordelene oppveiet ulempene. Men samtidig er det klart at det ikke var så mye et valg mellom goder, som det var et valg mellom onder. Mannen bak C++, Bjarne Stroustrup, skriver i (Stroustrup, 1991) at C++ er ment å være en ingeniørløsning, et kompromiss mellom fin teori og noe som ville bli akseptert i praksis.

Dette har ført til at språket har blitt populært, men også mer komplisert enn det kunne være, og bærer preg av å være laget i etapper. Først fordi C++ startet som en utvidelse av C, siden fordi stadig nye endringer og tillegg blir gjort. I skrivende stund er det ingen kompilatorer som implementerer språket slik det er beskrevet i (Stroustrup & Ellis, 1990).

3.5.2 Objekt-orientering

En teknikk som blir benyttet i programmet er objekt-orientering. Objekt-orientering er et ord som er veldig populært å bruke for tiden — omtrent som kunstig intelligens var et mote-ord for noen år siden.

Hva er det *egentlig* som skiller objekt-orientering fra andre teknikker? Det er flere elementer som er viktige.

- Objekt-orientering gir mulighet til å lage abstrakte data typer, med tilhørende operasjoner, som man siden kan opprette instanser av. Et eksempel kan være en ny type `Stack` med `push()` og `pop()` som operasjoner. Dette er viktig for å skjule detaljene i implementasjonen i moduler som bare skal bruke den nye typen. En `Stack` kan implementeres som en liste eller som et array, uten at andre rutiner har noe med det å gjøre.

  Denne egenskapen i et språk er ikke alene nok for å kalle det for objekt-orientert. Abstrakte data typer er velkjent fra "strukturert programmering".

- Objekt-orientering gir mulighet til å lage nye klasser som endrer eller legger til nye egenskaper i forhold til tidligere definerte klasser.

  Dette kalles nedarving (eng. *inheritance*) og er viktig for effektiv gjenbruk av kode.

- En slik ny type som er nedarvet fra en annen kan fremdeles brukes av funksjoner som forventer å få originalen. Dette er en kraftig mekanisme de stedene det passer å benytte seg av den.

Det kan være nyttig å skille mellom objekt-orientert programmering og objekt-orientert design. Det jeg har beskrevet til nå har vært egenskaper til språk for å understøtte objekt-orientert programmering. Objekt-orientert design er mer en måte å se på programmet på. Et program består av objekter (gjerne modelert etter objekter i den virkelige verden) som sender meldinger til hverandre. Man må ikke ha et objekt-orientert språk for å drive verken objekt-orientert design eller -programmering, men det gjør programmering enklere hvis språket har egenskaper som beskrevet over.

For å sammenfatte fordelene og ulempene med objekt-orientering, følger de kort her:

\+ Lettere å få til gjenbruk av kode.

\+ Lettere å vedlikeholde (velskrevet) kode.

\+ Lettere å lage abstraksjoner.

÷ Det kan bli endel ekstra kode å skrive hvis man skal gjøre det "riktig".

÷ Litt høyere inngangsterskel.

Konklusjonen er at objekt-orientering har en litt høyere terskel, men det er nærmest nødvendig ved store programsystemer. Et av kravene til programmet var at koden skulle være enkel å vedlikeholde, og objekt-orientering kan spille inn positivt her. Nå kan det diskuteres om dette prosjektet er stort nok til at fordelene ved objekt-orientering oppveier den forlengede læringstiden, men nettopp det å finne svaret på dette er noe av målet med oppgaven.

3.5.3 Bruk av InterViews

Vi skal representere transistorer i forskjellige retninger med ledere imellom. Da blir en ASCII-tegning fort uoversiktelig. Arbeidsmiljøet består av X-terminaler og arbeidstasjoner med høyoppløselige grafiske skjermer som alle benytter vindusystemet X11.

Dette gjør at det er naturlig å benytte disse mulighetene til å presentere utlegget grafisk. Gjerne med bruk av farger på de av skjermene som støtter det. Dessverre er det tungvint å skrive korrekte X11 programmer, så det er viktig å gjøre denne prosessen så enkel som mulig. Det beste ville være å finne et bibliotek som kunne gjøre grafikkprograme ringen enkel, men uten at man mister muligheter som ligger i systemet. Etter å ha undersøkt de aktuelle alternativene valgte jeg å benytte biblioteket InterViews.

InterViews står for **Inter**active **Views**, og er et C++ bibliotek som gir programmereren tilgang til X11. Selv om X11 er implementert i C (som ikke støtter objekt-orientering direkte), er det skrevet utifra en objekt-orientert tankegang. Derfor er det naturlig å kommunisere med det gjennom et objekt-orientert språk som C++, og et objekt-orientert bibliotek som InterViews. Det viste seg også at en programmerer slipper lettere unna ved bruk av InterViews. Det finnes mange ferdiglagde klasser som kan brukes direkte. Dette er alt fra brukergrensesnitt-elementer som menyer, knapper, rammer og scrollbar, til klasser for å håndtere kommunikasjon med brukeren, lese resurser og å håndtere generelle tegnevinduer. Hvis en klasse ikke passer helt, kan man lage en subklasse av den, som endrer akkurat den delen som ikke passet.

På denne måten får man effektiv gjenbruk av kode. Noe tilsvarende klarer man ikke med rutiner i et tradisjonelt prosedyre-orientert bibliotek. Hvis en rutine ikke passer må man skrive om hele rutinen. Man kan ikke beholde de delene av rutinen som var bra.

En viktig grunn til at InterViews ble valgt, er at det er et miljø av folk som bruker det, både her på instituttet, og ute i den store verden. Blant annet blir InterViews brukt til kommersielle produkter av flere firmaer. Det finnes en egen elektronisk nyhetsgruppe `comp.windows.interviews` som er viet diskusjon omkring Interviews. Tegneprogrammet `idraw` er et eksempel på en applikasjon som er skrevet i InterViews.

Det var relativt greit å tilpasse programmet til bruk av InterViews (hvis vi ser bort ifra den tiden det tok å finne den programmeringsfeilen som hadde sneket seg inn). Vanligvis er det en relativt lang opplæringstid. Dette ble noe enklere for meg fordi jeg ikke gjør bruk av så store deler av det InterViews har å tilby av knapper o.l. Jeg oppretter bare et tegnevindu som jeg så kontrollerer selv. I tillegg sjekker jeg om brukeren trykker på en av musknappene eller bruker tastaturet. Utifra hva brukeren gjør, vil jeg f.eks. vise neste løsning eller avslutte.

Et problemet var å skjønne at den vanlige event-løkken ikke kunne brukes i dette tilfellet. Det vanlige er at et InterViews-program er event-drevet (*event* betyr *hendelse*, men jeg fortsetter å bruke ordet event siden det er mer innarbeidet og dessuten klinger det bedre i mine ører), dvs. at kontrollen stort sett går i en løkke og leser event, og behandler dem. F.eks. hvis brukeren trykker ned en musknapp vil programmet kalle på den funksjonen som behandler det eventet. Den funksjonen kan da f.eks. beregne og vise frem neste lovlige utlegg. I dette tilfellet passer ikke denne måten å oppføre seg på, fordi det ikke finnes en liten grei funksjon som gir neste løsning. Isteden har vi en stor funksjon som genererer alle løsningene og som kaller på en utskriftsrutine. Det jeg gjorde var å la utskriftsrutinen som blir kalt hver gang det er en ny løsning, lese event slik at kontrollen stort sett ligger å venter på event i utskriftrutinen.

3.5.4 Testing

Det er menneskelig å feile, og det ser man tydelig når store programmer skal utvikles. Det er svært ofte mer eller mindre alvorlige feil i større programmer. Det hadde være fint om man automatisk kunne kontrollere om et program følger spesifikasjonen. Selv om man kan komme et stykke ved hjelp slike verifikasjonsmetoder, er det et faktum at en stor del av

tiltroen til programmer oppnåes ved testing. Siden dette er en viktig del av det å skrive korrekte programmer, er det lurt å legge litt ekstra arbeid i testing, slik at flere feil blir funnet på et tidligere tidspunkt.

Hvis man gjør det som er vanlig, nemlig teste litt overfladisk for hver kompilering, og å rette feil ettersom man oppdager dem, er det sannsynlig at flere feil ville ha blitt avdekket ved en mer grundig kontroll. Problemet er at man ikke orker å sjekke alt like grundig hver gang, så hvis man overlatter dette til å bli gjort for hånd, er det sjelden at de mer spesielle feilsituasjonene blir undersøkt. Når man til slutt støter på feilen, vet man ikke når man introduserte den.

For å øke sjansen for å finne en feil, og for å lette arbeidet med å teste pakken, bruker jeg det som kalles regresjonstest der hvor det er hensiktsmessig. Dette innebærer at man "fryser" de resultatene man forventer seg, og lagrer dem. Siden kan man automatisk sjekke om man fremdeles får de samme resultatene. På denne måten unngår man at det blir kjedelig, og det er lettere å gjennomføre store tester. Dette er en ide som kommer fra Software Enginering, og som burde vært mye vanligere enn den er.

I tillegg er det viktig å sjekke ting under veis. Hvis man inkluderer den standard headerfilen `assert.h` vil man få tilgang til en makro `assert(arg)`. Den vil sjekke om argumentet evalueres til FALSE (altså 0). Hvis det gjør det, vil programmet stoppe med en feilmelding. Dette er en grei måte å legge inn f.eks. sjekk av array-grenser. Jeg har et system for feilutskrifter som gjør at jeg kan variere hvor detaljerte utskrifter jeg vil ha ut i fra en variabel som gis verdi på kompileringstidspunktet. Tilsvarende med feilsjekkingskode. Noe jeg ikke har gjennomført helt enda, men som jeg har tenkt å begynne med, er at alle objektene skal ha en medlemsfunksjon `OK()`, som skal teste om det virker som objektet er i en lovlig tilstand. Den skal returnere TRUE dersom den er i en lovlig tilstand, og FALSE dersom den oppdager noe galt. Disse funksjonene kan jeg så kalle på siden, f.eks. ved å si `assert(obj.OK());`

Slike ting er spesielt viktige i et språk som C++ som ikke har noe apparat for innlagte tester under kjøring. Bare en ting som sjekk mot array-grenser ville ha funnet *mange* feil, og ikke bare for nybegynnere.

3.5.5 Abstraksjon

Hvordan kan vi bruke mulighetene C++ gir oss til å gjøre programmet enklere å forstå? Det gjelder å lage noen abstraksjoner som gjør det enkelt å tenke på problemet uten å henge seg opp i detaljene i programmeringen. Noen slike mekanismer vi har innført vil bli beskrevet i de følgende underavsnittene.

Klassen Undo

Dette er en klasse som gir programmereren et kraftig hjelpemiddel til bruk ved backtracking. Å lage prosedyrer som fjerner elementer man allerede har plassert ut er kjedelig, og dermed er det en kilde til intrikate feil. Hvis man bruker Undo-klassen ved alle endringer i datastrukturen trenger man ikke å lage slike "fjerneprosedyrer".

Et eksempel på bruken av klassen vil kunne se slik ut:

```
#include "backtrack.h"

void exampleroutine() {
  Undo Saver;                                   // Declare Saver as a new Undo-object
```

```
  int var1 = 1;
  int var2 = 2;

  Saver[var1] = -1;                    // Remember old value, and set var to -1
  Saver[var2];                         // Remember old value
  var2 = 75;                           // Set var2 to 75

  Saver.restore();                     // Restoring var1 = 1, and var2 = 2!
}
```

En komplikasjon er at man må overføre et Undo objekt til alle rutiner som kan tenkes å endre en variabel som skal innbefattes av systemet. Dessuten er den ikke tilpasset deallokering av minne. Dermed kan den ikke brukes til dynamiske strukturer uten at man stadig bruker mer og mer minne.

En kompilatorfeil Under arbeidet med implementasjonen av Undo kom jeg over en feil i C++-kompilatoren. Den dukker opp når man referanse-overfører en peker (hvis f.eks. parameteren er spesifisert som type `int* & var`). Adressen som blir plukket opp når man sier `&var` er ikke adressen til int-pekeren som ble sendt inn til prosedyren slik det skulle ha vært, men isteden adressen til den lokale variabelen som holder verdien midlertidig.

Etter at jeg fant feilen (noe som tok en del tid), var det relativt enkelt å omgå den, selv om det ble litt mindre elegant enn det ellers ville ha vært. I nyere versjoner av kompilatoren er feilen rettet, men det var etter at jeg var ferdig med den delen av koden, så jeg har ikke gått inn og endret dette tilbake til slik det skulle ha vært.

Funksjonen `try_next()`

Motivasjon for `try_next()` er å gjøre det lett å implementere algoritmer av typen "for alle objektene, prøv alle posisjonene helt til det blir funnet en tilordning av objekter til posisjoner som går igjennom alle testene". Hvis man skal gjøre dette "for hånd" blir programmet lett rotete, fordi mye av koden gjør ting som ikke driver algoritmen fremover. Hvis man bruker `try_next()`, vil den ta seg av det rekursive søket. Den benytter seg av Undo-klassen (som beskrevet over) til å ta seg av backtracking.

`try_next()`-rutinen hjelper oss med å komme opp på et høyere abstraksjonsnivå. Det er nærmest ekvivalent med å ha et nytt språk med nye konstruksjoner innebygd. Det eneste programmereren trenger å tenke på er hvordan selve avskjæringsrutinene skal oppføre seg — ikke hvordan man skal rydde opp etter seg o.l.

Klassene Pos og Elem Dette er to abstrakte klasser som definerer grensesnittet mellom try_next() og den faktiske implementasjonen. "Abstrakte klasser" er et C++ begrep som betyr at det ikke er lov til å opprette instanser av klassen. Man må lage subklasser av Pos og Elem for å få brukt `try_next()`.

Selve koden Siden funksjonen ikke er spesielt stor, skal jeg ta den med her:

```
void try_next(Pos &p, Elem &e, void (*finished)())
{
  Undo Saver;
```

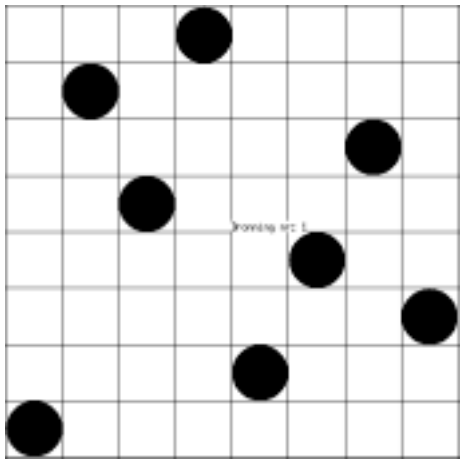
```
 for (; p.legal_pos(); p++) {                    // For all positions...
  if (andtests(p, e, Saver)) {                   // Try next pos if FALSE
   if (e.more_elem())                            // Last Elem?
    try_next(*p.next(), *e.next(), finished);    // ... then place next
   else
    finished();                                  // All elements placed successfully
  }
  Saver.restore();                               // Tried all positions, remove placement
 }
}
```

`andtests()` er en funksjon som prøver alle avskjæringene i rekkefølge, og som returnerer FALSE bare dersom en av testene gjør det.

`try_next()` er generell `try_next()` er en generell rutine som kan brukes som et hjelpemiddel for å implementere alle algoritmer som trenger backtracking av denne typen. For å demonstrere dette, og for å teste rutinen, brukte jeg den til å implementere en løsning på dronningproblemet.

Den generelle utgaven av dette velkjente problemet består i finne alle plasseringer av n dronninger på et $n * n$ sjakkbrett slik at ingen av dronningene kan slå hverandre.

Dette delprosjektet var vellykket, og `try_next()` var egnet som et hjelpemiddel for å løse dronning-problemet også. Jeg laget også et InterViews-grensesnitt til dette programmet, som kopien av skjermbildet i figur 3.6 viser.



Figur 3.6: Den første løsningen som blir funnet av mitt dronningoppgaveprogram som bruker `try_next()`.

Enkelt å prøve ut nye avskjæringer Ved bruk av `try_next()` blir det et skille mellom koden som har direkte med backtracking å gjøre, og koden som hører til den konkrete algoritmen. Dermed blir det enklere å prøve ut nye avskjæringer, siden man ikke blir fristet til å endre de grunnleggende rutinene.

Så noe av tankene er at `try_next()` skal gi en omgivelse der det er enklere å eksperimentere med nye avskjæringer.

For å se dette holdt i praksis , gjorde jeg en endring i dronningprogrammet slik at neste rad ikke blir "denne rad + 1", men isteden blir den raden som ligger på motsatt side av midtlinjen

(men slik at vi stadig nærmer oss midten). Tanken bak dette er å fange opp de tilfellene som ikke kan bli til en løsning på et tidligere stadium.

Motivasjonen for å teste ut dette var altså å se hvor store forandringer som skulle til for å implementere det. Nå var det riktignok ikke en spesielt stor endring, men det gikk uansett svært greit. En tilsvarende endring av et program som ikke bruker `try_next()` vil nok ofte medføre større problemer.

Bare for å ha tatt med resultatet av testen også, så følger det nå:

	$n = 4$	$n = 5$	$n = 6$	$n = 7$
Ordinær versjon	60	220	894	3584
Endret versjon	92	380	1434	6272

Tallene som står indikerer antall ganger testene har blitt kalt på, med de forskjellige verdiene av n (størrelsen på brettet/antall dronninger) og med de to versjonene av programmet. Hvis den nye algoritmen hadde vært god, skulle man forventet at man fikk færre tester med den endrede versjonen enn for den ordinære versjonen. Dette holdt ikke stikk for denne endringen, så den er nok ikke særlig lur. Poenget er ihvertfall at det var enkelt å gjøre denne endringen, og det tyder på at `try_next()` er et nyttig hjelpemiddel til å lage backtrackingprogrammer.

Iterasjon

Iterasjon er noe som trengs i alle programmer. I vanlige prosedyre-orienterte språk som C blir det fort store mengder med kode som ser noe slikt ut:

```
int x, y;
for (x = 0; x < XLIMIT; x++)
  for (y = 0; y < YLIMIT; y++)
    routine(x, y);
```

Selve koden sier ikke her noe om hva x og y er, og hvorfor vi skal gå i løkke. Det som egentlig skjer her, kan være:

```
Position p;
foreach (p ∈ Position)
  routine(p);
```

Nå er ikke foreach noen lovlig C++ konstruksjon, og det finnes ikke noen mulighet i C++ til å lage nye kontrollstrukturer heller. Altså får vi klare oss med det nest beste:

```
for (Position p; p.legal_element(); p++)
  routine(p);
```

Dette er relativt forståelig etter at man blir vant til tenkemåten, samtidig som det ser ut som C++ kode. Det *er* lovlig C++ hvis man på forhånd har laget klassen Position med passelig definisjon av medlemsfunksjonene `legal_element()`, konstruksjonsfunksjonen (dvs. den funksjonen som blir kalt på når objektet opprettes) og operatoren ++.

3.5.6 Datastrukturen i programmet

Datastrukturen er ofte det viktigste i et program. Den bestemmer hvor enkelt og effektivt det er å implementere de forskjellige algoritmene det kan være aktuelt å bruke.

Representasjon av utplasseringsområdet

Programmet bruker en matrise til å representere det området hvor man får lov til å plassere transistorer og ledere. Hvert punkt i denne matrisen er av typen Geometry. Et problem som dukket opp i forbindelse med klassen Geometry er hvordan man skal representere hva som ligger i et bestemt punkt. Her går jeg igjennom noen metoder.

- Ha et boolesk array hvor man kan slå opp på element-typen, og finne ut om det elementet er tilstede eller ikke.

  + Man trenger relativt få element-typer.
  + Dette er en grei representasjon.

  ÷ Det er tregt å finne ut hva som er i punktet. Selv om det er tomt må man gå igjennom hele arrayet for å se hva som ligger der.

- Slå element-typene sammen slik at det er et en-til-en forhold mellom lovlige kombinasjoner og tilhørende symbol. I så fall trenger man bare å lagre symbolet for å vite hva som er der.

  + Det er raskt å finne ut hva som er der, og om det er tomt.

  ÷ Det blir mange symboler å holde rede på.
  ÷ Det er vanskelig å finne ut om et bestemt (del-)element er tilstede.

- Ha et array, men pakk det sammen slik at elementene som er tilstede ligger i starten. Ha en teller som sier hvor mange elementer som er tilstede.

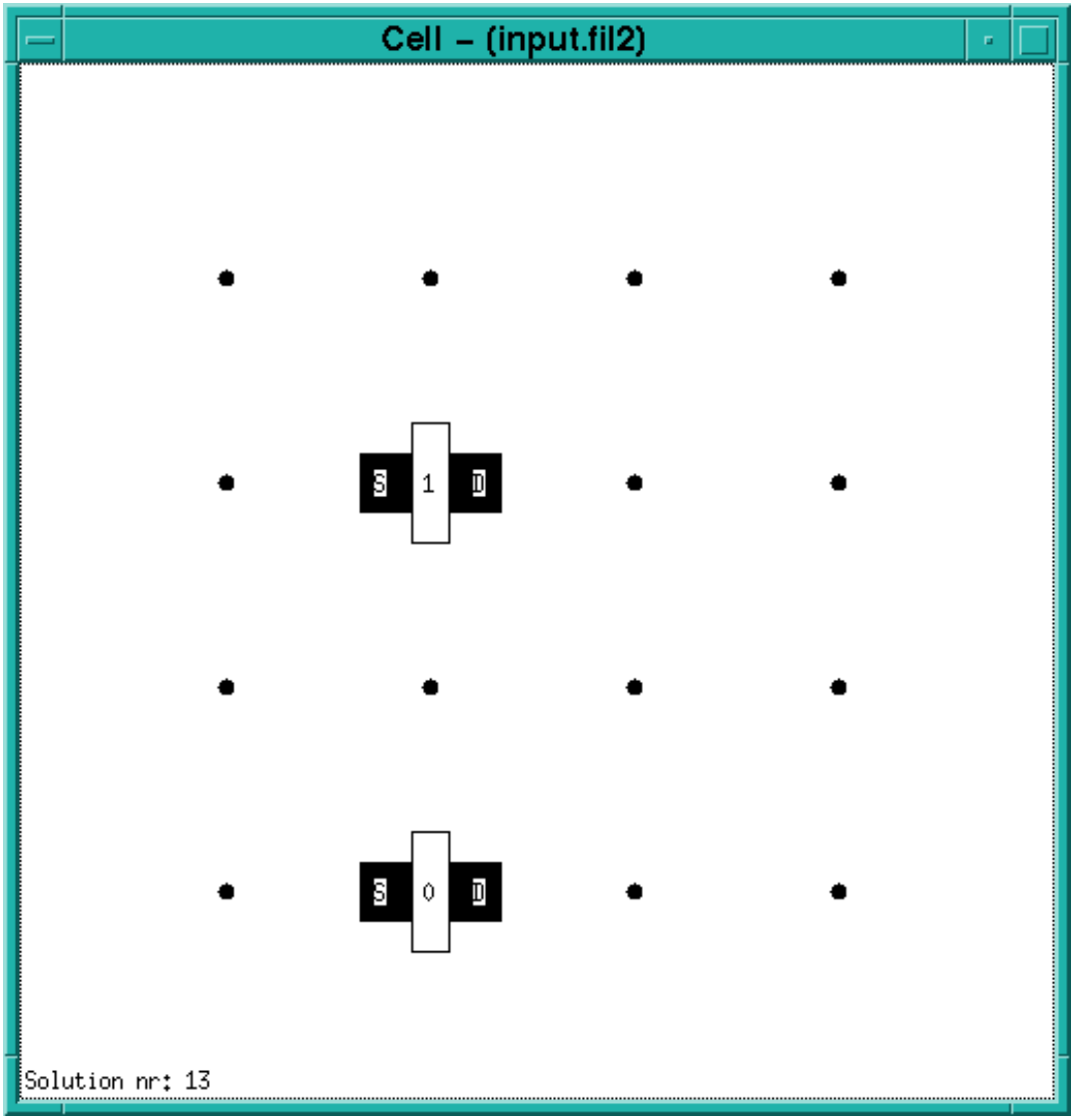
  + Man trenger relativt få element-typer.
  + Det er relativt enkelt å finne ut om et spesielt element er tilstede.

  ÷ Litt vanskeligere å sette inn og ut.
  ÷ Tar litt lenger tid å finne ut nøyaktig hva som ligger der enn hvis man lagrer alt i ett kombinert symbol.

Av disse metodene er det den siste som skiller seg ut som den mest fordelaktige, så det er den som er implementert.

3.5.7 Presentasjon av utlegg

Som nevnt i avsnitt 3.5.3 valgte jeg å bruke biblioteket InterViews til å lage et X11 grensesnitt til programmet mitt. I figur 3.7 ser vi et skjermbilde, etter at programmet har funnet posisjoner for transistorene.



Figur 3.7: Plasseringen er bestemt av mitt program. Kopi fra InterViews-vinduet.

3.6 Ruting i minimoduler

3.6.1 Håndtering av ruting

I det originale programmet til Arisland var noe av rutingen påbegynt. Jeg startet med de ideene han hadde, nemlig å gjøre ruting i samme stil som plasseringen. Dvs. å lage en algoritme som løser problemet optimalt, ved å benytte backtracking.

Problemet med en slik fremgangsmåte, er å få det effektivt nok. Derfor forsøkte jeg å finne en måte å avskjære suboptimale løsninger så tidlig som mulig, uten at man går glipp av de optimale løsningene.

Det å lage et program med mange slike optimaliseringer er tidkrevende, så for å få noe på lufta, valgte jeg å se på eksisterende rutere, for å se om jeg kunne koble meg opp mot dem.

3.6.2 Krav til lokalrutere

Krav til en lokalruter for at jeg kan bruke den:

- Tilgjengelig ved instituttet, helst med kildekoden.
- Håndterer koblingsboks — ikke bare kanaler.
- Terminaler inne i rutingarealet.
- Greit inn- og ut-format.
- Helst et variabelt antall lag, men minimum 2.
- Helst "flytende" kanaler ved alle fire sider.
- Rimelig effektiv.

3.6.3 Mighty var mest aktuell

Den eneste lokalruteren som var aktuell var Mighty. Se avsnitt 2.6.6 for en nærmere beskrivelse av algoritmene som brukes. Programmet Mighty er anerkjent som en god lokalruter for koblingsbokser. Den er grid-basert, og opererer med 2 lag som er holdt av til ruting. Programmet er ikke begrenset av et rektangulært ruting-område, men kan jobbe på områder som er avgrenset av rektilineære polygoner. Dessuten kan det håndtere terminaler inne i rutingområdet i tillegg til det vanlige, terminaler på kantene. Dette siste er en forutsetning for at Mighty skal kunne brukes fra mitt program. Programmet er på ca. 11 000 linjer med C-kode, og finnes tilgjengelig på instituttets maskiner.

3.6.4 Grensesnitt mot Mighty

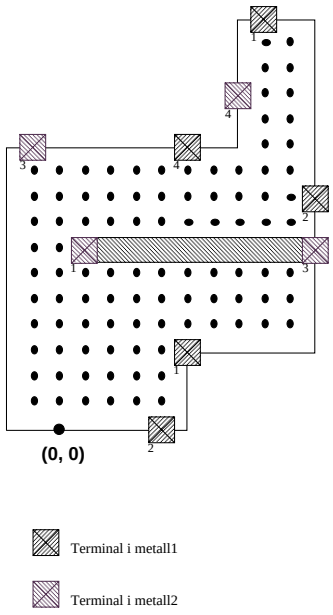
Mighty leser inn en fil som kan inneholde følgende opplysninger:

`number_of_nets` *nn*	Antall nett som skal kobles sammen.
`rectagoncorners` *h*	Antall hjørner i det rektilineære polygonet. som definerer omkretsen til rutingarealet.
$x_1\ y_1$	
⋮	Koordinatene for hjørnene.
$x_h\ y_h$	
`number_of_pins` *n*	Antall terminaler som skal sammenkobles.
$nett_1\ x_1\ y_1\ lag_1$	
⋮	Nett-nummer, koordinater og lag.
$nett_n\ x_n\ y_n\ lag_n$	
`left_list` *l*	Antall nett som skal kobles til venstre kant.
$nett_1$	
⋮	Nett-nummer.
$nett_l$	
`right_list` *r*	Antall nett som skal kobles til høyre kant.
$nett_1$	
⋮	Nett-nummer.

$nett_r$	
`sensitive_nets` s	Kritiske nett som skal rutes først
$nett_1$	
⋮	Nett-nummer.
$nett_s$	
`obstacles` o	Antall hindringer i rutingarealet.
$x1_1\ y1_1\ x2_1\ y2_1\ lag_1$	
⋮	Nett-nummer, koordinater og lag.
$x1_o\ y1_o\ x2_o\ y2_o\ lag_o$	

Se figur 3.8 for et eksempel på en innfil med tilhørende grafisk visualisering.

```
number_of_nets 4
rectagoncorners 8
-2 0
5 0
5 3
10 3
10 16
7 16
7 11
-2 11
number_of_pins 9
2 4 0 1
1 5 3 1
3 10 7 2
2 10 9 1
1 8 16 1
4 7 13 2
4 5 11 1
3 -1 11 2
1 1 7 1
left_list 2
1 3
sensitive_nets 1
3
obstacles 1
5 10 8 10 2
```



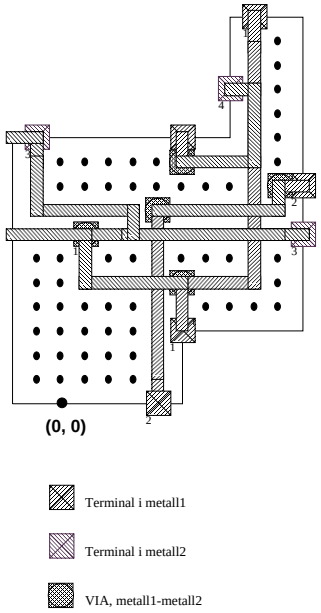
Figur 3.8: Innfil og tilhørende grafisk visualisering før Mighty blir kalt på.

Resultatet som Mighty leverer er en utfil som har følgende elementer:

`channelwiring`	Nøkkelord — start på data.
`vias` v	Antall via-kontakter mellom lag.
$x_1\ y_1\ nett_1\ lag1_1\ lag2_1$	
⋮	via med koordinater og lag-nummer.
$x_v\ y_v\ nett_v\ lag1_v\ lag2_v$	
`wires` w	Antall leder-segmenter.
$nett_1\ lag_1\ x1_1\ y1_1\ x2_1\ y2_1$	
⋮	Leder med nett, start/sluttkoordinater og lag.
$nett_w\ lag_1\ x1_w\ y1_w\ x2_w\ y2_w$	

Resultatet av innfilen over vil bli utfil og visualisering (kombinert med innfilen) som i figur 3.9.

```
channelwiring
vias 5
1 5 5 1 2
1 1 7 1 2
2 4 8 1 2
2 9 9 1 2
4 5 10 1 2
wires 23
2 9 9 10 9 1
2 4 0 4 1 1
2 4 8 9 8 2
2 9 8 9 9 2
2 4 1 4 8 1
1 8 15 8 16 1
1 8 5 8 15 1
1 5 5 8 5 1
1 5 3 5 5 1
1 1 5 5 5 2
1 1 5 1 7 2
1 -2 7 1 7 2
3 -1 10 -1 11 2
3 9 7 10 7 2
3 3 7 3 8 2
3 3 7 9 7 2
3 -1 8 3 8 2
3 -1 8 -1 10 2
3 -2 10 -1 10 2
4 5 10 5 11 1
4 7 13 8 13 2
4 5 10 8 10 2
4 8 10 8 13 2
```



Figur 3.9: Utfil og utseende etter at Mighty har gjort jobben sin.

Skriving av innfilen til Mighty

Mitt program har en representasjon av utlegget, mens Mighty har en annen. En likhet mellom de to er at begge bruker et grid, isteden for å plassere utleggselementene (med det mener jeg transistorer, ledere, kontakter, osv.) i vilkårlige punkter i planet. Hvis mitt program hadde brukt en slik friere representasjon, ville det ha vært veldig vanskelig å tilpasse det til et grid.

Mighty setter gjerne en via rett over en terminal. Dette er aldri ønskelig hvis terminalen er gate-noden i en transistor, men også uønsket for source/drain-nodene (men litt mer avhengig av designreglene).

Problemer med Mighty

Kompatibilitetsproblemer Selv om Mighty har mange gode egenskaper, så er ikke alt bare positivt. Mighty er regnet for å være bra, men det er tross alt på et litt annet område enn det jeg driver med. Mighty's styrke er i en omgivelse med moduler som skal kobles sammen, og ruting som foregår i kanaler og koblingsbokser. Jeg skal koble sammen transistorer, og rutingen foregår på den ledige plassen omkring/over dem. Derfor blir det endel kompatibilitetsproblemer.

- Mighty regner med at den skal løse relativt store instanser, mens jeg kommer til å gi den relativt små. Så små at problemet sannsynligvis kunne løses optimalt, eller ihvertfall nærmere optimalt enn det Mighty gjør nå. Alternativt kunne man oppnådd det samme raskere.

- Mitt program vil kalle på Mighty i mange tilfeller hvor det ikke lar seg gjøre å finne noe svar. Sannsynligvis vil Mighty forsøke ganske iherdig å finne en løsning. En ruter som i litt større grad var justert mot å stoppe hvis det ikke var plass, ville nok passet bedre inn i omgivelsene.

- Mighty støtter "flytende" terminaler, men bare på høyre og venstre side. Det blir et problem å få til flytende terminaler også på de andre to sidene.

- Mighty opererer med et fast bestemt sett med designregler. Dette er et problem, siden det er et mål for resten av systemet å være fleksibel med hensyn til dette, jfr. avsnitt 3.4.4.

- Mighty bruker bare to lag til ruting. Dette kan sees på som en variant av forrige punkt, men dette er mer spesifikt fordi resten av programmet har arbeidet ut i fra en teknologistil med to metall-lag tilgjengelig. Siden vi også til en viss grad kan bruke poly-laget til ruting, ønsker vi oss derfor at Mighty skal kunne benytte minst tre lag til ruting. For å komme rundt dette, kunne man ha utvidet Mighty til å bruke tre rutinglag, slik som (Sun, 1989).

- Mighty har ikke noe forhold til at den ikke kan sette en via rett over en transistor, siden den ikke vet hvor transistorene står. Dette er normalt ikke lov i noen alminnelige designregler.

For å hindre at Mighty lager ulovlige løsninger på grunn av det siste punktet, er det to muligheter:

Halvveis løsning Ved å si at det er en hindring i M2 rett over en transistor får jeg et lovlig utlegg, men mister noen løsninger.

Løsning: Endre Mighty slik at den tar en liste over punkter hvor det ikke er lov å sette viakontakter. Jeg har kikket på kildekoden for Mighty. Jeg fant ikke noe opplagt, men programmet er stort. Jeg har også skrevet et elektronisk brev til forfatterne av Mighty og fortalte om dette (og andre) problemer, men har ikke fått noe svar.

Generelle problemer Jeg har testet Mighty endel, og funnet noen svakheter.

I visse tilfeller (på bestemte inndata) stopper den ikke i det hele tatt. Isteden går den i evig løkke, og skriver ut linjer av typen:

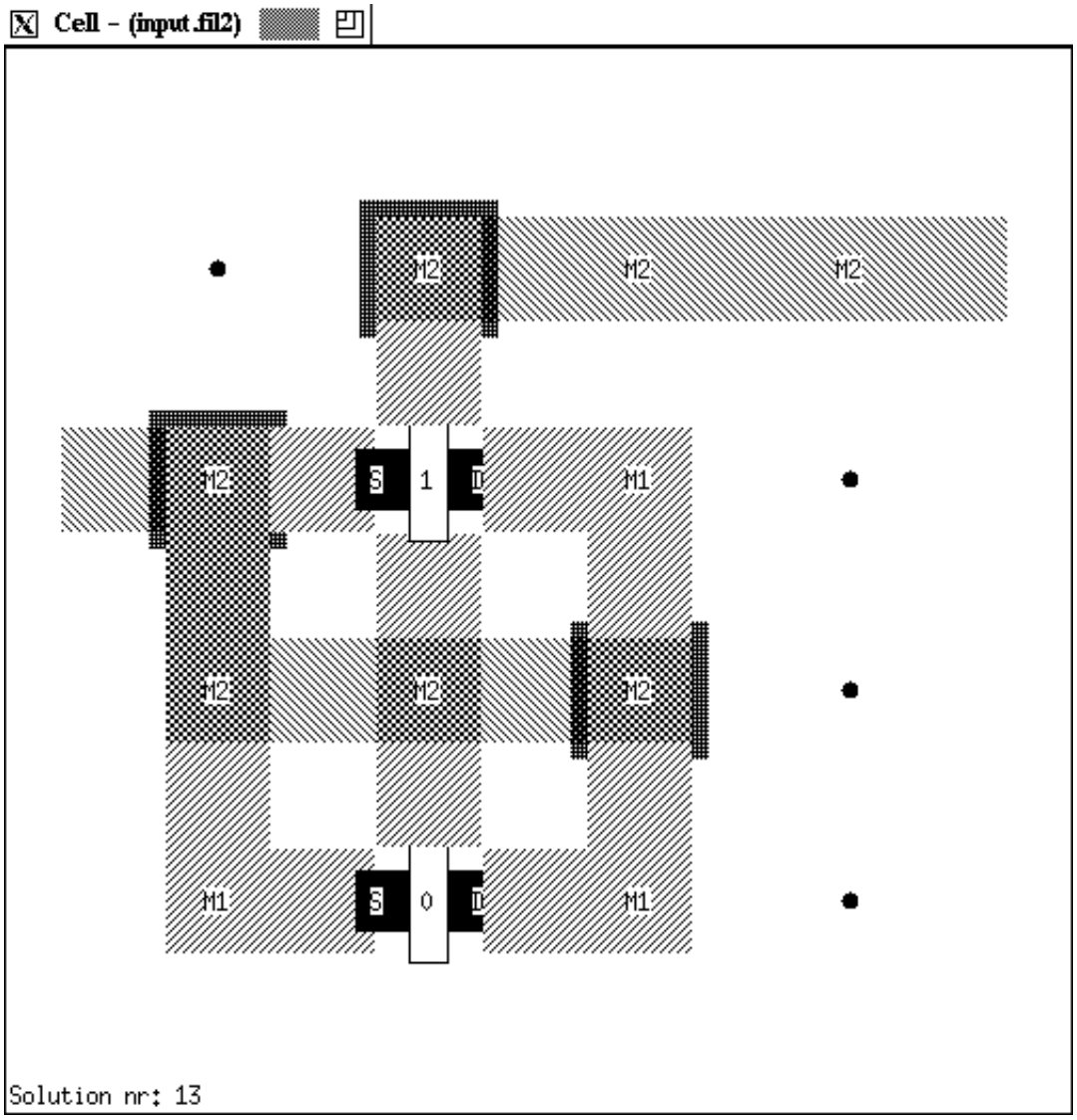
```
*** power improve called : net 2
*** power improve called : net 2
*** power improve called : net 2
```

Dette er svært uheldig, for det gjør at mitt program blir liggende å vente på at Mighty skal avslutte. En mulighet for å rette på dette er å gå inn i kildekoden til Mighty, og legge inn en sjekk på hvor mange ganger programet har vært på dette punktet i koden. Hvis det er mer enn f.eks. 20 ganger, avsluttes Mighty med en feilstatus.

En annen ting er at den ikke kobler alle terminalene sammen hvis to ligger oppå hverandre. Denne egenskapen er ikke dokumentert, og skaper litt problemer for mitt program. Det innebærer at konverteringen til Mightys format blir noe mer komplisert.

Presentasjon av utlegget

En viktig del av arbeidet med å samkjøre mitt program med Mighty har vært å kunne vise frem resultatet av en kjøring. Det har jeg løst ved å lese dataene fra Mighty og tilbake inn i mitt program, for så å vise dem fram med det InterViews-grensesnittet jeg allerede har laget. Dette blir seende ut som i figur 3.10 på neste side.



Figur 3.10: En ferdig rutet celle: Plasseringen er bestemt av mitt program, og selve rutingen er gjort av Mighty. Kopi fra InterViews-vinduet.

Da måtte programmet naturligvis utvides slik at de nye tingene kommer med. Det var relativt greit, siden jeg hadde regnet med å måtte utvide det når jeg laget det første gang. Det vanskeligste var å representere ledere, siden de ikke har utgangspunkt i et grid-punkt, men derimot ligger de imellom grid-punktene. Dette løste jeg ved å innføre fire nye arrayer i typen Geometry, et for hver retning. De sier hva som ligger i den retningen, f.eks. WIRE_M1.

3.7 Utvikling av et kriterium for partisjonering

En viktig del av oppgaven har vært å utvikle et partisjonskriterium, med basis i litteraturstudiene.

3.7.1 Vurdering av kriterier for bipartisjonering

Når det gjelder hvilket kriterium man skal bruke, er det klart at bipartisjonering vilkårlig spesifiserer et krav som algoritmen bruker mye ressurser på å oppfylle, samtidig som det ikke er strengt nødvendig for anvendelsen i dette tilfellet. Jeg tenker her på at antall noder i hver del blir forhåndsbestemt.

Samtidig er maksimal flyt, minimum kutt-kriteriet for lite opptatt av balanserte fordelinger. Derfor vil dette kriteriet ikke kunne brukes direkte på problemer innenfor VLSI.

Den metoden som det har blitt forsket mest på i det siste, har vært forholdskuttkriteriet, kombinert med spektrale metoder. Dette har gitt gode resultater i forhold til tidligere arbeider. En viktig grunn til det, tror jeg er at de løser det riktige problemet.

3.7.2 Vurdering av kriterier for multipartisjonering

Mange av de kriteriene som er foreslått her har kostnadsfunksjoner som vektlegger å finne færre, men større grupper, og gjerne med ganske ulik størrelse. Dette vil sannsynligvis reflektere kretsens oppbygning ganske godt, og det kan være ønskelig i mange sammenhenger. En slik sammenheng er hvis dette skal brukes som preprosessering før kjøring av en min-kutt- eller forholdskutt-algoritme. I min sammenheng vil det altså ikke være brukbart, siden Cell ikke kan legge ut grupper større en viss grense.

Mange av kriteriene tar ikke hensyn til hypergrafer, og jeg har ikke sett noen helhetlig behandling av to typer transistorer. Dette er ting som burde med i kriteriene, og ikke bare behandles ad hoc i algoritmene.

3.7.3 Overordnede krav ved utforming av et kriterium

Som vi har sett i gjennomgangen av den eksisterende litteraturen er det foreslått mange forskjellige kriterier for partisjonering/gruppering. Et kriterium vil bestemme hva algoritmene vi lager skal forsøke å oppnå. Siden vi vet hvilket faktisk problem vi skal løse kan det virke som om utviklingen av et passelig kriterium bare er et trivielt stykke arbeid — endel formalisering som må gjøres, men ikke egentlig noe vanskelig. Dette er ikke riktig, fordi det faktiske problemet vi skal løse er ofte så komplekst at det er vanskelig å formalisere.

Idealsituasjonen

I dette tilfellet er det faktiske problemet å finne den oppdelingen som fører til det beste ferdige utlegget. For å formalisere dette må vi bestemme nøyaktig;

- hvilke faser vi skal ha,
- hvordan grensesnittet mellom fasene skal være.

Dessuten krever denne problemformuleringen at en eventuell algoritme i praksis må løse hele utleggsproblemet for å finne en;

- optimal løsning.

Dette vil naturligvis ikke være aktuelt, for grunnen til at fasene ble innført var fordi hele problemet er for komplekst til å løses på en gang.

I praksis

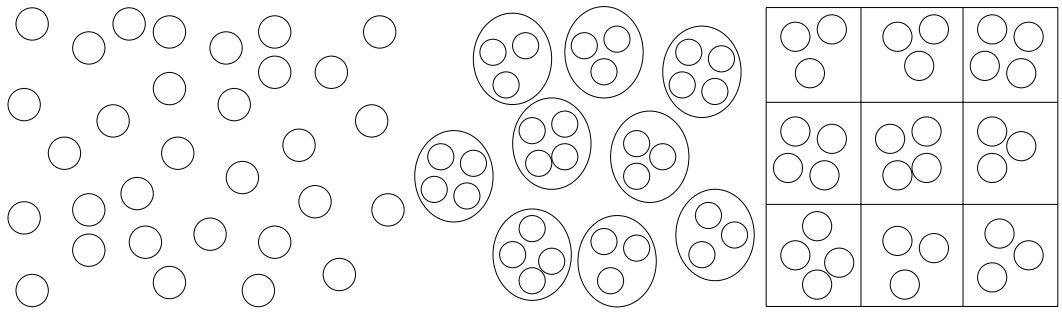
Et kriterium må for å være praktisk gjennomførbart, tilpasse seg;

- oppdeling i faser,
- være enkelt nok til at en effektiv algoritme kan utvikles,
- samtidig som det reflekterer alle de viktige sidene ved det virkelige problemet.

Ved å abstrahere vekk sider ved det virkelige problemet kan vi oppnå en rasjonaliseringsgevinst: man kan finne et kriterium som passer for flere praktiske problemer. Dette er selvsagt gunstig; disse "renere" problemene kan studeres da grundigere. Men hvis det er viktig med høy kvalitet på løsningene må man passe seg for å ikke overforenkle kriteriet ved å fjerne relevante sider av det virkelige problemet. Det fører til at algoritmen kan bruke mye ressurser på å optimalisere uvesentlige sider ved problemet, mens andre viktige sider blir overlatt til tilfeldighetene.

3.7.4 Det problematiske grensesnittet mellom partisjonering og plassering

Et viktig spørsmål er hvilke faser vi skal ha når det gjelder partisjonering og plassering, og hvordan fasene skal integreres. Skal vi dele problemet i to faser partisjonering og plassering, eller skal de kombineres i en fase? Figur 3.11 inneholder en skisse av disse fasene.



(a) (Nodene fra) grafen før partisjonering og/eller plassering.

(b) Grafen er partisjonert, men delene har ikke posisjoner.

(c) Grafen er delt opp og delene har posisjoner.

Figur 3.11: Skal man ha to faser, eller skal de kombineres til en fase?

Det er vanskelig å finne en algoritme som løser begge fasene på en gang, som tar hensyn til alt som er relevant, som er rask og gir gode resultater. Problemet er at plasseringen har mye med globalruting å gjøre, så i praksis må man nærmest kombinere tre faser. Dette gir en sterk indikasjon på at det kan være lurt å ha to separate faser.

Men hvis man velger å ha en egen partisjoneringsfase, får man et stort problem. Det blir mye mer definitivt at to transistorer havner i forskjellige grupper. Sett at man har flere transistorer som opplagt hører sammen, men det er litt flere enn det er plass til i en del. Hvis gruppene hadde hatt posisjoner ville man ha hatt mulighet til å legge transistorene i

nabogrupper. Forskjellen på at to transistorer er i samme gruppe eller i nabogrupper er ikke så stor.

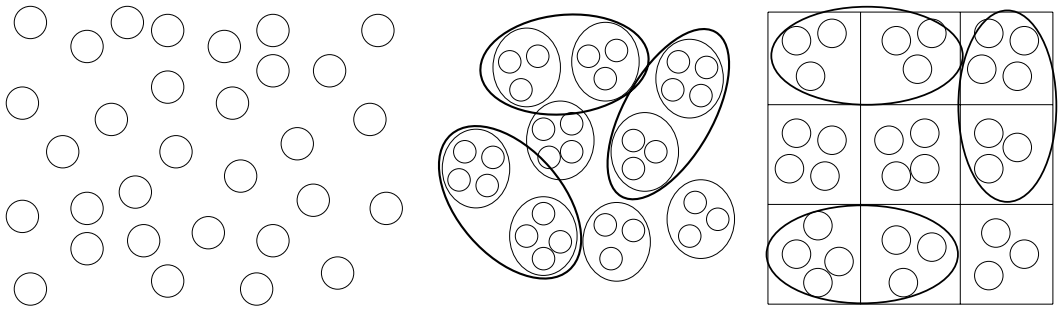
Dessuten kan man bare håpe at plasseringsdelen putter de to gruppene nær hverandre. Det er slett ikke sikkert at plasseringsalgoritmen får til dette. Det er delvis fordi plasseringsfasen er vanskelig i seg selv, og delvis fordi partisjoneringsfasen kan legge opp til motstridende krav som ikke alle kan oppfylles samtidig. Dette bunner i at partisjoneringsalgoritmen ikke kan vite hvor gruppene kommer til å bli plassert til slutt.

Hvis man har to faser, vil man sannsynligvis oppdage i plasseringsfasen at det er noen gruppesammensetninger som er litt uheldige, og som hadde kunnet vært gjort bedre når man vet hvilke grupper som havner ved siden av hverandre. En mulighet er da å begynne å flytte om på transistorene, men dette er tungvint (krever enda en fase), og vil sannsynligvis ikke hjelpe så veldig mye. Enten gjør man bare noen få forandringer som sannsynligvis ikke vil være verdt bryet, eller så må man gjøre store forandringer, og det blir mye jobb og vil gjøre mye av partisjoneringen unødvendig. For å unngå dette, må partisjoneringsfasen ha tilgang til mer informasjon om plasseringen av gruppene mens den arbeider, og da er vi jo på god vei tilbake til å bare ha en fase igjen.

3.7.5 Partisjonering med naboer

En mulig utvei på dilemmaet som er beskrevet i forrige avsnitt er følgende: isteden for å la partisjoneringsfasen ha fullstendig oversikt over plasseringen til gruppene, kan vi gjøre som Telenor — vi kan klassifisere avstander slik at to transistorer kan være i samme gruppe (lokaltakst), i nabogrupper (nabotakst) eller i helt forskjellige grupper (fjerntakst). De naboforholdene vi dermed genererer blir da hint til plasseringsprosedyren.

Naboskap er noe som hører med gruppene. Hvis en gruppe er nabo av en annen, vil også det omvendte gjelde. Naboskap er altså en refleksiv relasjon. Samtidig er det viktig at det ikke blir generert verken for mange naboskapsforhold (for alle gruppene kan jo ikke være naboer på en gang), eller for få (i så fall får man jo ingen glede av denne mekanismen). Dette er illustrert i figur 3.12.



(a) (Nodene fra) grafen før partisjonering og/eller plassering.

(b) Grafen er partisjonert. Delene har noen indikasjoner om naboforhold.

(c) Grafen er delt opp og delene har posisjoner. Naboene er forsøkt plassert ved siden av hverandre.

Figur 3.12: En ny mulighet: Midtfasen inkluderer informasjon om naboforhold!

Slik dette er tenkt brukt, har det to formål. Det ene er at to nabogrupper kan sees på som

en stor gruppe. Dette løser mange av problemene for partisjoneringsprogrammet, som nå kan operere med et forenklet avstandsmål. Det andre formålet er å la partisjoneringsprogrammet kunne kommunisere litt mer om hva som allerede er gjort, slik at plasseringsprogrammet slipper å finne ut dette om igjen.

Naboskapene er ment som hint om at disse to gruppene burde ligge kant til kant.

3.7.6 Hva plasseringsprogrammet skal gjøre med naboforhold

Vanligvis er ikke naboforhold en del av grensesnittet mellom partisjoneringsfasen og plasseringsfasen. Hva skal så plasseringsprogrammet gjøre med informasjonen om naboforhold?

Plasseringsprogrammet står fritt til å se totalt bort fra alt som har med naboforhold å gjøre. Det ligger ikke noe informasjonen der som ikke også ligger i oppdelingen og nettlisten. Hvis partisjoneringsprogrammet har gjort jobben sin vil man se at nabogrupper er sterkere forbundet enn andre grupper, men om dette ikke skulle stemme, er det nettlisten man skal tro på. Naboforholdene er kartet, mens nettlistene er terrenget. Dette er en god analogi, for naboforholdene fungerer omtrent som et på kart flere måter: litt grovere skala, men ofte enklere å finne fram med enn ved å lete rundt i terrenget/nettlisten.

Utifra det som blir skrevet over, virker det som om naboforhold egentlig ikke trenger å sendes over til plasseringsprogrammet, siden man kan finne omtrent det samme i nettlisten. Så hva er da det nye med å introdusere naboforhold? Foruten at vi sannsynligvis slipper mye dobbeltarbeid vil det virkelig nye med nabokonseptet være at partisjoneringsalgoritmen bruker dette som en del av sitt kriterium.

3.7.7 Krav til løsninger på det aktuelle problemet

Som nevnt er det foreslått mange kriterier for partisjonering og gruppering. Jeg skal her se hvilke momenter som må med i denne sammenhengen, hvis man skal unngå overforenkling. Det første kravet er at:

- Kriteriet skal regne med hypergrafer.

Dette betyr ikke at algoritmene ikke kan benytte vanlige grafer som simulerer hypergrafer hvis det er en heuristikk som gir gode løsninger på det egentlig problemet, men det er resultatet på hypergrafer som skal være målet for hvor god kvalitet løsningene har. Vi ønsker også å:

- Dele opp i et forhåndsbestemt, ganske høyt, antall deler.

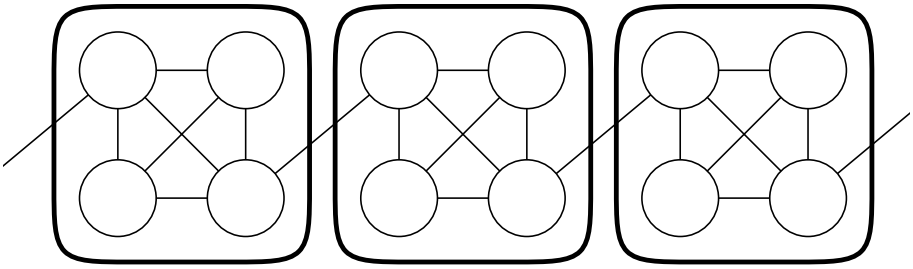
Grunnen til at antall deler skal være forhåndsbestemt er at vi senere ønsker å plassere delene utover et rektangel med visse krav til høyde/bredde-forholdet. Det hjelper ikke hvis 41 deler gir opphav til en "god" oppdeling, hvis det er 42 deler vi trenger for å plassere delene i et rektangel. Siden antall deler er forhåndsbestemt er det altså k-partisjonering, og ikke det mer generelle multipartisjoneringproblemet, vi skal se på.Hensynet til den neste fasen er også begrunnelsen for det neste kravet:

- Det skal være omtrent like mange noder i hver del, sannsynligvis rundt fem noder.

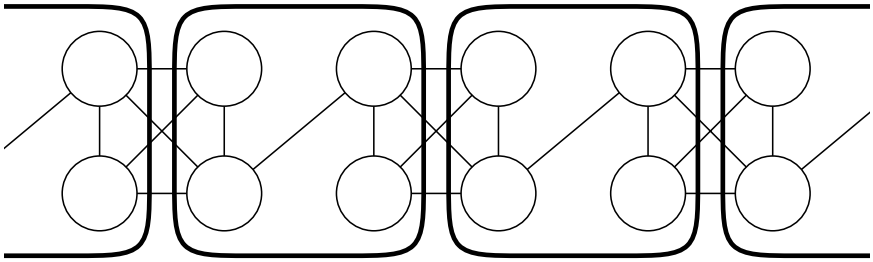
Siden programmet Cell fungerer best når den får et begrenset antall noder å arbeide med, hjelper det ikke at kretsens oppbygning gir opphav til en oppdeling med svært ulik størrelse på gruppene.

3.7.8 Gode og dårlige løsninger

Hittil har jeg bare beskrevet krav til lovlige løsninger, men ikke sagt noe om hva som skiller gode løsninger fra dårlige. Det vanlige er å si at en god løsning vil ha mye kommunikasjon internt i gruppene, og mindre eksternt. Egentlig gir ikke dette et helt riktig bilde av prosessen. Se på følgende figur (3.13):



(a) Denne "gode" oppdelingen har mye kommunikasjon internt og lite eksternt. . .



(b) . . . mens her er det omvendt.

Figur 3.13: Selv om oppdelingen i (a) virker mye bedre enn den i (b), kan de legges ut på samme måte.

Den første oppdelingen har mye kommunikasjon internt og lite eksternt, mens den andre har det omvendt. Likevel kan begge oppdelingene gi opphav til samme utlegg (nemlig slik som i figuren). Det som redder oppdelingen i (b) er at all ekstern kommunikasjon er så lokal at den lar seg håndtere ved å legge ut gruppene riktig. Løsningen i (a) vil fremdeles være god selv om man permuterer gruppene, mens det samme vil ikke være tilfelle i alternativ (b).

Hvis partisjoneringsalgoritmen ikke har noen garanti for hva plasseringsalgoritmen vil gjøre, vil (a) derfor være den beste løsningen. Normalt vil det derfor være grunn til å gjøre suboptimale valg andre steder, for å oppnå en slik god oppdeling her. Hvis man hadde hatt en slags garanti for at delene skulle plasseres slik som her, kunne man ha valgt en slik oppdeling, hvis det hadde ført til å kunne gjøre optimale valg andre steder.

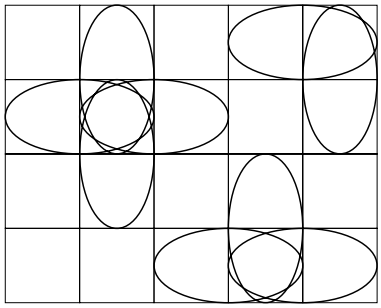
3.7.9 Hvordan oppnå gode løsninger

Garantien ønsket over, kommer ved å endre grensesnittet mellom partisjoneringsfasen og plasseringsfasen til å også ta hensyn til naboer, som nevnt i avsnitt 3.7.5. Jeg foreslår å gi partisjoneringsprogrammet litt kontroll med hvor delene sannsynligvis havner, og bruke

denne informasjonen i partisjoneringsfasen. Dette vil kunne gjøre både partisjoneringen og plasseringen enklere, og gjøre resultatet mer egnet som utgangspunkt for et utlegg.

Dette må altså være en del av kriteriet for hva partisjoneringsalgoritmen skal oppnå: den skal generere et passelig antall naboforhold, og kommunikasjon mellom naboer "koster" mindre enn kommunikasjon mellom deler som ikke er naboer.

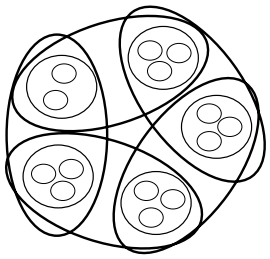
Det bør imidlertid ikke genereres naboskap som sannsynligvis ikke kan oppfylles, for det vil svekke poenget med å ha dem. Dette innebærer blant annet at ingen gruppe kan ha mer enn fire naboskap, og at ikke alle gruppene kan ha så mange (på grunn av kantene). Figur 3.14 viser hvordan dette henger sammen.



Figur 3.14: En gruppe kan ha to, tre eller fire, avhengig av om gruppa plasseres henholdsvis i hjørnet, langs kanten eller i midten av utlegget. Siden partisjoneringsprogrammet ikke vet hvor de havner, er det best å ikke generere for mange naboer.

I det hele tatt bør plasseringsprogrammet få endel spillerom til hvordan utlegget skal bli uten å måte bryte med naboskapshintene.

For at naboforholdene skal ha noen sjanse til å bli tatt hensyn til av en plasseringsalgoritme er det også andre begrensninger på hvordan de kan defineres. Naboforholdene i figur 3.15 er et eksempel på en nabospesifikasjon som ikke kan oppfylles slik at alle naboer ligger kant-til-kant.



Figur 3.15: Fem naboforholdene i sirkel som her kan ikke legges ut i en matrise, slik at alle fem gruppene er kant-til-kant med sine to naboer.

Alle naboforholdene kan representeres som en graf, med grupper som noder, og slik at det går en kant mellom to noder, dersom gruppene er naboer. For at føringen som naboforholdene representerer skal kunne bli tatt hensyn til, må denne nabografen være en partiell matrisegraf (eng. *partial grid graph*). En matrisegraf har en regulær struktur a la et ruteark:

nodene står i skjæringspunktene og kantene er strekene mellom skjæringspunktene. En partiell matrisegraf, er en matrisegraf som har fått fjernet noen kanter og eventuelt noen noder.

Dette kan partisjoneringsalgoritmen ta hensyn til for å sikre seg at naboforholdene er rimelige. Men partisjoneringsalgoritmen bør være mer restriktiv enn dette. Hvis man f.eks legger på alle naboforholdene som er lov etter denne metoden, vil også all plasseringen være spesifisert. Det vil i så fall si at enten må plasseringprogrammet forkaste endel naboforhold for å få litt handlefrihet, ellers så har vi egentlig laget et plasseringsprogram i stedet for et partisjoneringsprogram. Begge deler er uheldig.

Det fine med nabokonseptet er at det får en oppdeling som den i figur 3.13 (b) til å bli bra, dersom de riktige naboforholdene er angitt, og dersom partisjoneringsalgoritmen har en rimelig garanti for at naboforholdene vil bli respektert av plasseringsalgoritmen.

3.7.10 I/O-tilkoblinger med krav til plassering

I/O-tilkoblinger har ofte angitt hvilke sider av modulen de ligger på, og eventuelt også omtrent hvor på sidekanten (f.eks på nederste kant 30% mot venstre hjørne). Dette er informasjon som bør taes hensyn til i partisjoneringsfasen. Hvis ikke, risikerer man å bygge opp en minimodul som inneholder to terminaler som hører hjemme på hver sin side av det totale utleggsarealet.

En måte å unngå dette på, er å først gå igjennom terminalene, og se hvilke som bør inn i samme del, og hvilke som bør inn i nabodeler, og hvilke som ikke kan være i samme deler. Terminaler som bør sammen, kan man bare legge i samme del med en gang. Terminaler som bør være i nabopartisjoner, bør markeres så det blir tatt hensyn til. Nøyaktig hvordan dette blir gjort vil være avhengig av hvordan resten av algoritmen behandler slike naboskapsforhold. Terminaler som ikke bør være i samme del, kan legges inn i en sperreliste, slik at man kan teste i den hver gang man vurderer operasjoner som kan føre til at slike terminaler kan havne sammen.

3.7.11 Kritiske nett

Noen rutere kan ta imot lister med nett som er viktigere enn andre, og som derfor må prioriteres så de blir korte. Dersom nettlisten spesifiserer slike nett, bør dette taes hensyn til allerede i partisjoneringsfasen. Sålenge de kritiske nettene kobler sammen få elementer kan man forsøke å plassere dem i samme modul, eller i nabomoduler. Her er det på sin plass å sjekke mot sperrelisten nevnt over, så ingen slike krav blir brutt. Hvis man klarer å plassere elementene som de kritiske nettene kobler sammen nær hverandre (i samme del eller i nabodeler), vil en ruter sannsynligvis klare å lage et utlegg hvor de kritiske nettene blir korte.

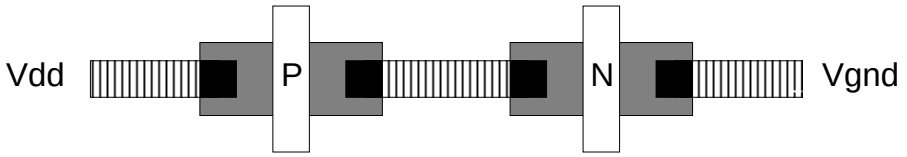
Dette gjør i grunnen bare partisjoneringen enklere, for på den måten blir endel av arbeidet gjort på forhånd. Resten av instansen vil være mindre enn hvis man ikke hadde fått oppgitt noen kritiske nett på forhånd.

3.7.12 Finne transistorer som kan settes tett sammen

Poenget med å la minimodulene bli lagt ut automatisk, er at totalutlegget skal bli kompakt. Men for at det skal kunne bli kompakt, må programmet som legger ut minimodulene ha det riktige materialet å arbeide med. Hvis man f.eks. skal få til sammenkobling av transistorene, må slike transistorer som har direkte forbindelser til hverandre havne i samme deler.

3.7.13 Finne P- og N-transistorer som hører sammen

I en nettliste beregnet på CMOS, vil det normalt være omtrent like mange N-transistorer som P-transistorer. Dette er avhengig av utleggsstilen, og gjelder for f.eks komplimentær CMOS og pass transistor logikk, men ikke for domino logikk. I de stilene hvor det gjelder, vil det også ofte forekomme par av P- og N-transistorene som hører sammen. Slike kalles heretter for tvillingtransistorer, og har form som i figur 3.16.



Figur 3.16: N- og P-transistorer som går mellom V_{dd} og V_{gnd} forekommer ofte, og bør ligge nær hverandre.

3.7.14 Formalisering av kriteriet

En mulig formell beskrivelse av kriteriet for gode partisjoner er som følger:

Instans: Hypergraf $H = (V, E)$, hvor V er en union av V_{term}, V_P, V_N. Heltall b, B, k og J.

Spørsmål: Kan man finne en k-veis partisjonering, P^k, som deler V opp i k disjunkte grupper $C_1, C_2, \ldots, C_k$ og en liste over naborelasjoner N, slik at for alle i mellom 1 og k vil:

$$(C_i = C_{i_{PN}} \cup C_{i_{term}}) \wedge (C_{i_{PN}} \subseteq V_P \vee C_{i_{PN}} \subseteq V_N) \wedge$$
$$C_{term} \subseteq V_{term} \wedge b \leq |C_{i_{PN}}| \leq B$$

og slik at $f(P^k, N) \leq J$?

I denne formelen uttrykker $C_i = C_{i_{PN}} \cup C_{i_{term}}$ at hver gruppe består av to deler, og $C_{i_{PN}} \subseteq V_P \vee C_{i_{PN}} \subseteq V_N$ utrykker at en av disse delene må inneholde noder fra enten V_P (P-transistorer) eller V_N (N-transistorer). Dette sikrer at en gruppe ikke kan inneholde både P- og N-transistorer. $C_{term} \subseteq V_{term}$ sier at de resterende nodene i en gruppe må være fra V_{term} (terminaler). Merk at C_{term} kan være tom. Det kan forsåvidt $C_{i_{PN}}$ være også, men b i kravet $b \leq |C_{i_{PN}}| \leq B$ vil som regel være større enn 0 og dermed sikre at hver gruppe består av minst b transistorer. B vil angi den øvre grensen for størrelsen på en gruppe.

N er en liste over par av grupper, og angir at gruppene i et slikt par er naboer.

$f(P^k, N)$ er en kostnadsfunksjon, som måler hvor god en løsning er i forhold til en annen. Den må ta hensyn til at kanter som går mellom grupper som er relatert i N, teller positivt i retning av en bedre oppdeling, på samme måte som kanter som går internt i en gruppe.

3.8 Partisjoneringsalgoritmer

3.8.1 Vurdering av algoritmer for bipartisjonering

Metoder basert på forbedringer av Kernighan-Lin algoritmen er ganske bra, og er gjerne standarden som andre algoritmer måles etter. Men det er mulig å oppnå bedre resultater.

Simulert størkning gir gjerne svært gode resultater, men bruker også lang tid. Hvis man skal implementere en enkel utgave, er det forholdsvis fort gjort, men å finjustere metoden og simuleringsparametrene for å få programmet til å gå fortere kan være en svært møysommelig oppgave. En av grunnene til at metoden er så populær, er at den kan tilpasses mange forskjellige kombinatoriske problemer.

Nevrale nettverk kan foreløpig ikke konkurrere med de beste algoritmene for grafpartisjonering, men det kan tenkes at de vil kunne gjøre seg mer gjeldende etterhvert.

Genetiske algoritmer faller i omtrent samme kategori som simulert størkning. Metodene som bruker egenvektorer og matriser gir gode resultater, men er begrenset til vanlige grafer.

3.8.2 Vurdering av algoritmer for multipartisjonering

Den eneste algoritmen som passer direkte inn i modellen med å lage mange deler med få noder i hver del er (Feo & Khellaf, 1987). Dessverre fungerer denne algoritmen for hypergrafer. De andre målene og algoritmene kan nok til en viss grad justeres til dette med større eller mindre vansker.

Av algoritmene som fungerer direkte på hypergrafer er det (Lee et al., 1993) som virker mest direkte overførbar til mine forhold siden det er lagt inn sperrer mot at gruppene skal bli for store. En ting jeg ikke helt liker med denne algoritmen er hvorfor noden som blir plukket ut til å bli assimilert i en gruppe blir det på grunnlag av forbindelsene den har til alle nodene som allerede er plassert. Det vil ofte skje at en annen node har sterkere forbindelser til spesifikke grupper. Kanskje det hadde vært bedre å velge node og gruppe samtidig slik at noden klaffet enda bedre med den gruppen den havner i? En slik endring vil føre til at algoritmen bruker noe lenger tid, men burde kunne føre til høyere kvalitet på løsningene.

3.8.3 Konstruktive og iterative algoritmer

Som nevnt i avsnitt 2.3.3, er heuristiske algoritmer gjerne av to typer, konstruktive og iterative. De konstruktive algoritmene bygger en løsning opp fra grunnen, mens de iterative forbedrer en eksisterende løsning. Man skulle kanskje tro at det beste ville være en kombinasjon, ved at man først bygger opp en løsning konstruktivt, som så blir forbedret. Dette blir for eksempel forfektet i (Lee et al., 1993), hvor det står:

> It should be pointed out that a better initial partitioning solution always leads to a better solution after iterative improvement.

Hvorvidt dette er riktig, kommer an på hvordan de to algoritmene utfyller hverandre.

Hvis begge algoritmene er sterke på samme "del" av problemet, vil den iterative algoritmen ikke klare å forbedre løsningen noe særlig utover det den konstruktive algoritmen leverer fra seg. Da vil programmet bare bli vanskeligere å implementere, og kreve lenger kjøretid, uten at man oppnår noen forbedring av betydning, i forhold til å bare bruke én av algoritmene. Hvis de to algoritmene er sterke på forskjellige forhold ved problemet (f.eks ved at den konstruktive algoritmen håndterer visse globale parametre godt, mens den iterative algoritmen ofte forbedrer lokale strukturer) vil en kombinasjon kunne føre til gode resultater. Dette er ihvertfall konklusjonen i (Johnson et al., 1989) etter å ha forsøkt å kjøre forskjellige iterative algoritmer etter hverandre.

Iterative algoritmer arbeider gjerne utifra lokal optimalisering, dvs. at løsningen ikke vil forandre seg mye fra runde til runde. Dette er greit for relativt små instanser, men når instansene blir svært store, vil det ta mange runder for å komme fra den opprinnelige løsningen til

en løsning nær det optimale. På veien henger algoritmen seg opp i for mange lokale detaljer til at den klarer å se de store linjene som skal til for å løse problemet virkelig godt.

Alle heuristiske iterative algoritmer er avhengige av den løsningen de får oppgitt når de starter. På grunn av dette, og på grunn av eventuelle vilkårlige valg som blir tatt underveis, vil resultatene bli forskjellige fra gang til gang. For å være rimelig sikker på at løsningen man har funnet ikke er unormalt dårlig, er det derfor vanlig å kjøre algoritmene noen ganger. Blant annet derfor er det vanligvis vanskelig å analysere hvor gode resultater man kan forvente å få.

Konstruktive algoritmer kan også være vanskelige å analysere, og kan også ha problemer med å gi samme svar hver gang. Men det er ihvertfall litt større sjanse til oppnå noe slikt med konstruktive algoritmer.

3.8.4 Partisjoneringsalgoritmen "Nabo"

Parametre til algoritmen

s ønsket størrelse på gruppene (vil være bestemt av hva som er mest egnet nettstørrelse for programmet Cell, sannsynligvis ca 5)

x bredden på utlegget målt i antall grupper

y høyden på utlegget målt i antall grupper

$k = x \times y$ antall grupper vi skal ha (vil være omtrent $|V|/s$). Vi kan også snakke om k_P og k_N, som er henholdsvis antall P-grupper og antall N-grupper vi skal ha.

Forarbeid

Finn initielle grupper ved å samle transistorer av samme type (P og N) som hører sammen. De hører sammen hvis de er forbundet av små nett (2–5 transistorer) med bare diff-til-diff og gate-til-gate forbindelser. Først finner man grupper med transistorer som er forbundet med diff-terminaler, fordi slike transistorer må i samme gruppe for at de skal kunne plasseres tett.

Her kommer det en delalgoritme for å samle slike par: Finn alle overganger fra diff til diff (mao source/drain til source/drain) mellom transistorer av samme type. Alle slike 2-terminal nett er kandidater for direkte sammenkobling av transistorene. Her er pseudokode for å finne slike par. Den kaller på rutinen `RememberPair()` som må tåle at et par blir lagt inn flere ganger.

```
foreach (n in nets) {
    if (n→size == 2) {
        t1=n→node[0];
        t2=n→node[1];
        if (t1→type == t2→type &&
            (t1→source == n || t1→drain == n) &&
            (t2→source == n || t2→drain == n)) {
        RememberPair(t, other);
    }
  }
}
```

Alle 2-, 3- og 4-terminal nett som forbinder transistorer av samme type og samme lag, er også kandidater for å dele gruppe.

Deretter tar man gate-forbindelser på tilsvarende måte. (Gate-terminaler ser ikke ut til å være forbundet på denne måten? Ihvertfall gjør ikke dette noen forskjell på de testdataene jeg har til rådighet.) Man må passe på at ingen grupper blir større enn s, siden de da ikke kan brukes til slutt.

Oversikt over materialet

Da har vi følgende materiale å arbeide med:

- I/O-terminaler (enten med fast/flytende posisjonsangivelse eller uten)
- P-grupper og N-grupper
- Enslige transistorer som ikke er plassert i noen gruppe
- Gruppene har muligheter for naboskapsforhold, men foreløpig vil det ikke være noen slike, så denne datastrukturen er tom.

Lage k grupper med riktig antall P- og N-grupper

Vi ønsker å ende opp med k grupper til slutt, og med et forhold mellom N- og P-grupper som tilsvarer forholdet mellom N- og P-transistorer. Normalt er det like mange transistorer av hver type, så normalt skal vi ha like mange grupper av hver type også. Sannsynligvis må vi nå justere antall grupper opp og/eller ned slik at disse kravene blir oppfylt.

Kombinere grupper Hvis det er flere grupper av en type enn det skal være til slutt, må noen av dem slåes sammen. Normalt gjøres sammenslåingen ved å velge 2 og 2 smågrupper slik at den kombinerte gruppa ikke blir større enn maksimalgrensen, s. Men vi kan risikere at alle gruppene er større enn $\frac{s}{2}$, og da kan vi ikke slå sammen noen uten at kombinasjonen blir større enn s. Dette vil sannsynligvis forekomme relativt sjelden, men hvis det skjer kan man flytte noen noder slik at en gruppe får maksimal størrelse og den andre blir mindre. Nå kan den minste kombineres med en annen gruppe (eventuelt ved å gjenta denne operasjonen noen ganger).

Det er beste å slå sammen grupper som har felles kanter eller som ligger inntil felles transistorer. En annen faktor som teller i favør av sammenslåing er hvis begge gruppene er naboer med en tredje gruppe.

Tvillingforhold skaper nabogrupper

Tvillingtransistorer bør ligge nær hverandre, så det er naturlig å plassere hver del av et slikt par i nabogrupper. Så hvis man finner et par som har en transistor i hver sin gruppe, bør disse gruppene settes til å være naboer.

Følgende pseudokode finner slike par ved å se på alle P-transistorer, men kunne like gjerne ha startet med alle N-transistorene isteden. Den ser finne hvilke transistorer som har source/gate forbindelser til V_{dd}/V_{gnd}, følger den motsatte ut-kanten (drain/source), og ser om den går til source/drain på en N-transistor via et nett med mer enn 2 noder. Hvis drain/source på N-transistoren går til V_{gnd}/V_{dd}, er de to transistorene "tvillinger" og huskes.

```
foreach (pt in P−transistors) {
  follow = NULL;
  if (pt→source == Vdd || pt→source == Vgnd) {
```

```
        follow = pt→drain;
    }
    if (pt→drain == Vdd || pt→drain == Vgnd) {
        follow = pt→source;
    }
    if (follow ≠ NULL) {
        foreach (nt in follow→net→translist) {
            if (nt→type == N) {
                if ((nt→source == Vdd || nt→source == Vgnd
                    || nt→drain == Vdd || nt→drain == Vgnd) {
                    RememberPair(pt, nt);
                }
            }
        }
    }
}
```

Dette kan nå brukes til å bygge opp gruppene med N-transistorer etter at gruppene med P-transistorer er funnet. Men det er nok bedre å gjøre dette simultant, siden det kan ligge verdifull informasjon om hvordan gruppene skal bygges opp i både P- og N-gruppene.

Siden det ikke er mulig for en gruppe å plasseres kant-til-kant med mer enn to grupper av forskjellig type, er det viktig å minimere antall påkrevde naboforhold. Det kan gjøres ved å samle flere slike tvillingtransistorer i grupper som er naboer. Informasjonen om slike tvillinger må altså brukes aktivt i prosessen med å slå sammen grupper og til assimilasjonsprosessen som legger nye transistorer inn i grupper.

Skape nye grupper Hvis det er færre grupper av en type enn det skal være til slutt, må noen nye grupper defineres. Dette gjøres ved å velge ut noen transistorer som skal danne grupper. Her kan det også være lurt å ta hensyn til I/O-terminaler, som er beskrevet i neste punkt.

I/O-terminaler

Grupper som inneholder I/O-terminaler vil sannsynligvis ligge langs kantene (eller i hjørnene) av utleggsarealet når plasseringsalgoritmen har gjort jobben sin. Derfor kan ikke slike grupper ha like mange naboskapsforhold som grupper som ligger nær midten av utleggsarealet. Dette er forklart nærmere i avsnitt 3.7.5. På grunn av dette, er det viktig at I/O-terminalene havner i gruppene på et tidlig tidspunkt, slik at det kan bli tatt hensyn til dette.

Det er også en annen faktor med disse terminalene som gjør at de bør inn i grupper tidlig. Som nevnt i avsnitt 3.7.10 er det viktig at I/O-terminaler ikke havner i samme gruppe hvis det er angitt posisjoner som tilsier at de skal stå et stykke unna hverandre. Dette håndteres med sperrelister som beregnes på forhånd. Hvis en I/O-terminal havner i en gruppe, vil gruppen arve sperrelisten, slik at de samme restriksjonene gjelder for hele gruppa.

Lage flere naboforhold

Det er uheldig hvis noen grupper ikke har noen naboer. I så fall undersøker man forbindelsene mellom slike naboløse grupper, og lager nye naborelasjoner hvis det er tilstrekkelig kommunikasjon (felles kanter osv) mellom dem. Denne måten å plukke ut kandidater til naboskap på, er lignende den som brukes for å finne kandidater til grupper som kan slåes sammen hvis

det er for mange grupper til å begynne med. En gruppe bør ikke ha mer enn to naboer av samme type, og bør sjelden ha mer enn tre naboer totalt.

Assimilasjon av enslige transistorer inn i grupper

Sist, men ikke minst, skal enslige transistorer assimileres inn i passelige grupper. Dette gjøres på følgende måte:

For hver gruppe som ikke er full, beregn verdien til alle enslige transistorer som gruppa har felles kant med og som er av samme type som gruppa. Verdien beregnes ut fra en formel som må bestemmes ved testing. Faktorer som skal være med i formelen er:

> Antall kanter som går mellom transistoren og gruppa og gruppas naboer teller positivt. Muligens skal små nett veier tyngre enn store. Verdien blir høyere dersom gruppa og nabogruppene har få noder. Antall kanter som går til andre grupper enn den aktuelle og nabogrupper teller i negativt. Kanter som går til andre enslige transistorer skal det sannsynligvis sees det bort fra.
>
> Hvis en gruppe har flere transistorer som er del av par av tvillingtransistorer, bør man prøve å få transistorene i den andre enden til å gå sammen i en gruppe. På denne måten vil P-gruppene påvirke N-gruppene, og motsatt.

Den kombinasjonen av gruppe og transistor som har høyest verdi blir plukket ut, og transistoren blir innlemmet i gruppa. Dette gjentar seg til alle transistorene er plassert.

3.8.5 Implementasjonen

Jeg har forsøkt å ta hensyn til en del forhold som er spesielt for VLSI-design, og dette har satt sitt preg på hvordan implementasjonen har blitt. Dette gjenspeiles både i datastrukturen og i algoritmen.

Siden jeg ønsker å håndtere hypergrafer må en nettliste representeres som en graf med objekter og pekere. En nabomatrise som det ofte er vanlig å bruke i partisjoneringsalgoritmer er ikke egnet i mitt tilfelle.

Jeg fant noen benchmark-data og et system for innlesing av disse. Det påvirket også hvordan implementasjonen ble. For eksempel var det nå naturlig å velge C som programmeringsspråk, fordi innlesningsystemet la opp til dette.

3.8.6 Testdata

Det er viktig å prøve algoritmene på realistiske nettlister. Dette vil gi en verdifull indikasjon om hvor gode de er, for ikke å snakke om at man ofte får nye ideer ved å se hvordan ting fungerer i praksis.

Jeg har hentet noen testdata som tidligere ble vedlikeholdt på MCNC, men som nå blir vedlikeholdt ved “CAD Benchmarking Lab (CBL)” ved “N C State University”.

Systemet for innlesning av testdataene består av en “front end” som leser dataene og sjekker at de er syntaktisk korrekte. Denne er skrevet i YACC (Yet Another Compiler Compiler) og C. Dette blir så koblet sammen med en “back end” som jeg har skrevet.

Testdataene er beskrevet i VPNR som er et kretsbeskrivelsesspråk som er orientert mot plassering og ruting.

De kretsene som jeg har brukt til testing er følgende:

Navn	I/O-terminaler	P-transistorer	N-transistorer	Totalt	Nett
ti_alu	22	153	153	328	169
accum	19	208	208	435	205
fract	24	345	377	746	385
struct	64	4495	4495	9054	4529

Som vi ser er alle disse større enn de 100–200 transistorene som jeg har som målsetning å fordele i grupper.

På testdataene er det ikke lagt inn så mye krav til plassering av I/O-terminaler. Derfor har sperrelister (som nevnt i avsnitt 3.7.10) ikke blitt implementert i denne omgang.

3.8.7 Oppsummering

Siden min algoritme tar hensyn til mange VLSI-spesifikke detaljer, er ikke min implementasjon av algoritmen "nabo" en generell grafpartisjoneringsalgoritme. Man kunne tenke seg at den kunne brukes til det, men i konkurranse med slike algoritmer ville den nok ikke gjøre det særlig skarpt. Dessuten er det ikke så mange generelle algoritmer som er egnet til k-veis partisjonering av hypergrafer

Siden det ikke finnes noe lignende program er det vanskelig å teste hvor godt dette programmet gjør det i forhold til andre program. Resten av verktøyet må også implementeres før det er mulig å si så mye om hvor godt programmet er.

Imidlertid har programmet vært verdifullt for å teste ut de ideene jeg har hatt, og jeg tror det har hjulpet meg til å få bedre føling med hva det kan lønne seg å vektlegge i utviklingen av partisjoneringsalgoritmer innefor VLSI.

3.9 Kernighan-Lin

3.9.1 Implementasjon av KL

Jeg har implementert KL algoritmen for å få en føling med hvor god den er. Først laget jeg en prototype i språket Perl, og siden implementerte jeg det i C. Grunnen til at det ble skrevet i C og ikke i C++, er først og fremst av hensyn til å enkelt kunne lese testdataene inn i programmet. Prototypen håndterte bare vanlige grafer, mens den ferdige versjonen fungerer også for hypergrafer.

3.9.2 Resultat av å kjøre KL

Når jeg testet KL på testdataene (spesifikt ti_alu) fant jeg ut noe interessant. Jeg foretok 10 kjøringer hvor jeg delte opp grafen tilfeldig i to partisjoner. Maksimalt ble 134 nett kuttet, minimalt ble 119 nett kuttet, og gjennomsnittet var 125.2. Ved å kjøre KL på disse oppdelingene varierte kuttverdiene fra 23 til 35, med et gjennomsnitt på 28.7.

Det interessante at jeg også sjekket hva kuttverdien var når den første halvparten av nodene som ble lest inn ble plassert i en partisjon, og den halvparten som ble lest inn ble plassert i den andre. Den var nemlig 29 i utgangspunktet, som ble forbedret til 17, ved å kjøre KL. I løpet av kjøringen ble det stort sett flyttet over terminaler (som blir lest inn sist). Dessuten hadde man ikke trengt noe så avansert som KL, for hele forbedringen kom i starten av den første runden.

Gjennomsnittet etter å ha kjørt KL var altså bare såvidt bedre enn den opplagte kuttverdien som innlesningen gir oss. Dette forteller oss at KL ikke er så god som en kunne håpe. Det

gir også en svært god indikasjon på at nettlisten ikke var tilfeldig satt opp i den fila som ble lest inn. På dette grunnlaget vil den beste partisjoneringsalgoritmen bestå i å ikke kaste den informasjonen som ligger i den opprinnelige rekkefølgen i nettlisten. Dette blir selvsagt litt ad hoc, men det har en dypere melding til oss.

Hvis dette er typisk for nettlister, betyr det at grensesnittet mot de tidligere lagene er feil. Den informasjonen burde ikke bare taes hensyn til hvis den tilfeldigvis er der, men man burde kreve at den skulle være der. Altså at den logiske syntesen kommuniserer mer til partisjoneringsfasen enn det som er vanlig.

3.10 Plassering av minimoduler

Plasseringsproblemet i denne oppgaven er enklere enn for mange andre systemer siden alle modulene er omtrent like (som behandlet i avsnitt 2.4). Fordi problemet er såpass forenklet, er det fristende å ikke behandle plasseringen som en egen fase, men heller slå den sammen med en annen.

En mulighet er å slå plasseringsfasen sammen med partisjoneringsfasen. Et problem med dette, er at plasseringsproblemet har mye å gjøre med rutingen. Hvis vi ikke skal prøve å takle alle disse fasene på en gang, partisjonering, plassering og globalruting, må man ha en metode for plasseringsalgoritmen for å estimere hvilke plasseringer ruteren vil sette mest pris på. Her er det to veier å gå. Vi kan forsøke å estimere lederlengdene som ruteren vil lage, slik at vi kan finne plasseringer som minimerer dette. Det er gjort mange forsøk på dette, men det er erfaringsmessig vanskelig å komme med gode estimater (Lengauer, 1990, avsnitt 7.1.1.3). Den andre veien å gå, er å sørge for at ingen punkter i utlegget vil være overbelastet med signaler. Dette kalles mettningskontroll og vil vanligvis innebære at man gjør noe tilsvarende partisjonering. (Mayrhofer & Lauther, 1990) er en artikkel som nettopp går denne veien, og det kan virke som en lovende metode. En ulempe er at "top-down"-algoritmen som er brukt må ta hensyn til mange faktorer på en gang, og vil sannsynligvis få problemer med å skalere opp til større utlegg.

En annen mulighet er å forsøke å slå sammen plasseringsfasen og globalrutingsfasen. Dette kan integreres med det som er gjort i denne oppgaven. Dette vil gi mettningskontroll (pga grupperingen som blir gjort), samtidig som plasseringsalgoritmen har en relativt enkel oppgave. Litteraturen som behandler dette blir beskrevet i (Lengauer, 1990, avsnitt 8.8).

En strategi for plassering som jeg synes virker lovende, er å konsentrere seg om å plassere nettene, isteden for å se hovedsaklig på hvor minimodulene skal stå. Hvis man får plassert nettene jevnt utover utleggsarealet, kan man være ganske sikker på at minimodulene blir plassert slik at det ikke blir forstoppelse i rutingfasen. Dermed er det sannsynlig at utlegget blir enkelt å rute.

3.11 Ruting mellom minimoduler

3.11.1 Globalruting mellom minimoduler

Hvis man ønsker å integrere globalruteren tettere til resten av systemet, kan følgende forslag undersøkes. Ved å foreta globalruting før man har lagt ut minimodulene, har man muligheten til å spesialbehandle kritiske nett. Hvis et nett blir avgjort å være viktig nok (eller hvis veldig mange ledere ønsker å krysse samme område) kan man velge å la et (eller flere) nett gå tvers

over en minimodul. Dette krever at minimodulene kan ta hensyn til tidligere plasserte elementer, og dette er tilfellet i denne oppgaven. Hvis det er plassert elementer i utleggsarealet, kan man risikere at området minimodulen har til rådighet blir for lite, og i så fall er man nødt til å utvide området.

Alternativt kan man legge ut minimodulene først. Hvis man undersøker minimodulen, vil man kunne finne ut om det er rom for å legge en eller flere ledere over minimodulen. Ved å gjøre det på denne måten, vil man aldri trenge å utvide arealet til minimodulene. Dessverre vil det ikke så ofte være mulig å legge ledere over minimodulene heller, siden det ikke blir eksplisitt satt av plass.

Begge disse metodene krever at globalruteren blir skrevet spesielt for å dra fordel av denne muligheten. Men det vil ikke kreve noe omfattende samarbeid mellom modulene. Det er mulig å lage andre varianter over dette temaet, som integrerer de to fasene enda tettere. For eksempel kan man la globalruteren be programmet som legger ut minimoduler om å forsøke å få plass til en ekstra leder på tvers av rektangelet, men ikke utvide arealet dersom det ikke går. Hvis utleggsprogrammet kan svare "nei", må globalruteren skrives så det blir tatt høyde for det. Det er vanskelig å si hva som kommer til å bli best av disse forskjellige forslagene, så det beste er å avgjøre det ved videre arbeid og testing. Ihvertfall gir fleksibiliteten ved å kunne arbeide med et område hvor noe av plassen allerede er brukt, muligheter som man ellers ikke har.

3.11.2 Lokalruting mellom minimoduler

Lokalrutingen kan gjøres etter vanlig mønster, men også her har vi muligheten til å gjøre en vri. Hvis to moduler skal settes ved siden av hverandre, og det normalt ville blitt en kanal i mellom, kan man isteden benytte følgende strategi.

Man legger ut den første minimodulen. Området som blir satt av til den andre minimodulen plasseres slik at det overlapper området til den første. Dermed blir kanalrutingen integrert med utlegget av minimoduler. Istedenfor å betrakte minimodulenes sidekanter som rette streker, vil de bli behandlet individuelt av programmet som legger ut minimoduler.

Dette vil sannsynligvis føre til tettere utlegg, samtidig som man slipper å implementere en egen kanalruter. En mulig ulempe er at det kan vise seg å ta lenger tid enn ved spesialiserte lokalrutere. Dessuten bør man nok ha en egen koblingsboks-ruter for mere generelle tilfeller.

Kapittel 4

Drøfting av resultater

4.1 Betydningen av kriterievalg

Det er skrevet mange artikler innenfor dette feltet. Noe av grunnen til det, er at det blir gjort små, inkrementelle forbedringer av tidligere arbeider. Ved å justere noen parametre (og kanskje ved å implementere programet på en rask maskin), får man en algoritme som arbeider litt raskere enn tidligere. Dette er selvsagt verdifullt, men det er enda viktigere med arbeider hvor det blir presentert noe genuint nytt. Hvis man bare gjør inkrementelle forbedringer, går man kanskje glipp av de store forbedringene som en ny struktur ville kunne ha gitt.

Et eksempel på dette, er at det er arbeidet mye med algoritmer for min-kutt. Når så forholdskuttkriteriet ble presentert, ble det fort klart at for mange anvendelser er dette et mye riktigere mål. Det er rett og slett nærmere det vi egentlig ønsker. Samtidig er det svært ofte k-partisjonering vi egentlig ønsker å utføre, mens det er bipartisjonering som blir studert mest. Det er greit å studere bipartisjonering, siden det kan gi innsikt i en viktig del av hvorfor problemet er vanskelig, men jeg tror at større del av innsatsen skulle vært rettet mot å dele direkte opp i flere deler. I den siste tiden *har* det kommet flere artikler om dette, men jeg tror det fremdeles er rom for forbedring.

I litteraturen er det en tendens til at utviklingen av algoritmer er mye viktigere enn utvikling av kriterier. Jeg mener at det burde legges mye større vekt på valget av kriterier enn det vanligvis gjøres i dag. For eksempel blir mange av grupperingsalgoritmene brukt innenfor CMOS teknologi. Men det er få, såvidt meg bekjent, som har arbeidet med grupperingsalgoritmer som tar hensyn til at P- og N-transistorer skal i forskjellige grupper. Jeg kjenner heller ikke til noe formalisert kriterium som tar hensyn til de to typene transistorer.

4.2 Evaluering av partisjoneringsalgoritmer

Ideelt vil vi teste algoritmene på instanser som er typiske for de som algoritmene skal brukes på, og se hvor stort avvik vi får fra det optimale. Dessverre er partisjoneringsproblemet så vanskelig at det kan være vanskelig å finne ut hva den optimale verdien skal være. Da har vi to muligheter:

- Vi kan sammenligne med andre algoritmer.
- Vi kan lage instansene slik at vi vet hva den optimale verdien er.

Den første måten krever at vi har andre algoritmer å sammenligne med. For at sammenligningen skal bli meningsfyllt må de andre algoritmene løse nøyaktig det samme problemet.

Hvis kriteriene som brukes er like, men ikke helt identiske, er det litt usikkert hva man får ut av en direkte sammenligning.

Metoden med å lage spesielle instanser hvor man vet hva den optimale verdien er, har sine egne problemer. For at det skal være noe poeng med å bruke disse instansene må de ligne på "ekte" instanser. For at dette skal være en interessant metode, må problemet med å lage slike spesielle instanser la seg løse rimelig effektivt.

I artikkelen (Krishnamurthy, 1987) er det beskrevet en metode for å lage slike instanser til bipartisjonering av hypergrafer med min-kutt-kriteriet, og k-partisjonering av vanlige grafer. Store deler av oppbygningen av grafene kan styres slik at grafene får mange statistiske egenskaper fra "ekte" grafer. Det er nok allikevel vanskelig å gjøre dette på en skikkelig måte hvis man har et litt spesielt kriterium (f.eks med naboer).

De færreste har lagt kravet om at P- og N-transistorer skal i forskjellige grupper inn i kriteriet på samme måte som jeg gjør. Det blir heller ordnet litt ad hoc. Og nabo-konseptet er også så vidt jeg vet helt unikt. Derfor er det ikke sikkert at en sammenligning med andre algoritmer vil være helt meningsfylt.

Jeg har hentet ned et sett med testdata, og kikket litt på dem. Vi får se om jeg får tid til å bruke dem på noen algoritme.

4.3 Evaluering av programmet Cell

Det er ønskelig å forsøke å finne ut hvor bra dette programmet er. Her skal jeg se litt på hvilke alternativer vi har for å evaluere kvaliteten på programmet. Med kvalitet mener jeg både kvaliteten på løsningene, og på hvor lang tid som blir brukt for å finne dem.

Dessverre er det vanskelig å måle disse sidene ved programmet. For å ta tidsaspektet først, så er det vanskelig å si noe om det fordi programmet bruker en backtrackingsstruktur. Det er en metode som i verste tilfelle blir eksponentiell, men som bruker avskjæringer som får den gjennomsnittlige kjøretiden til å bli mye lavere enn det. Derfor er O-notasjon et litt uinteressant mål for kompleksiteten til algoritmen. Samtidig er det ikke noen generelle metoder for å finne gjennomsnittlig ytelse. Dette er kort og godt et vanskelig problem som må undersøkes spesielt for hver algoritme.

Kvalitetsaspektet er også vanskelig å måle. Et problem er at programmet arbeider i en kontekst, der programmet blir kalt opp til å løse små delproblemer, som siden skal bli satt sammen til en komplett løsning. Uten denne konteksten er det vanskelig å si om noe er bra eller dårlig.

Siden algoritmen arbeider ut i fra prinsippet om å prøve alle muligheter, vil den — dersom den er implementert riktig — alltid gi optimal kvalitet på løsningene.

For å si om noe er bra eller dårlig, må vi ha noe å sammenligne med. I dette tilfellet er det vanskelig å sammenligne med det optimale, på grunn av den spesielle konteksten programmet arbeider i.

Hvis man ikke kan sammenligne med det optimale, er det vanlig å sammenligne med konkurrerende programmer. I dette tilfellet finnes det ingen konkurrerende programmer, siden dette er en ny angrepsvinkel.

Det at det er vanskelig å teste hvor bra denne metoden er kan jo tilsynelatende se ut som en svakhet, for hvis hele verktøyet hadde vært ferdig kunne man ha testet det på virkelige kretser og sammenlignet resultatet med det andre fullstendige verktøy gir. Imidlertid var hele verktøyet tenkt som et stort prosjekt der det bl.a inngikk en doktorgrad, og det var aldri meningen at denne hovedoppgaven skulle ta for seg mer enn en liten del av dette.

program	transistorer	plassering	løsninger	tidsforbruk
c_all	1	36	12	0.0
c_illegal	1	36	12	0.0
c_diff	1	36	12	0.0
c_bare	1	36	36	0.0
c_all	2	468	56	0.0
c_illegal	2	468	144	0.0
c_diff	2	468	144	0.0
c_bare	2	1332	1296	0.0
c_all	3	2484	96	0.0
c_illegal	3	5652	1728	0.1
c_diff	3	5652	1728	0.0
c_bare	3	47988	46656	0.3
c_all	4	5940	0	0.1
c_illegal	4	67860	20736	1.0
c_diff	4	67860	20736	0.5
c_bare	4	1727604	1679616	14.7
c_all	5	5940	0	0.1
c_illegal	5	814356	248832	15.2
c_diff	5	814356	248832	7.0
c_bare	5	62193780	60466176	465.7

Tabell 4.1: Resultater av kjøring av Cell med forskjellige avskjæringer på utleggsareal med størrelse 4×4 punkter.

I virkeligheten tyder vanskene med å finne lignende program-moduler å koble mitt arbeid sammen med, på at det her er tenkt en del nye tanker som det ville være interessant å forfølge videre for å se om de fører frem.

4.4 Kvaliteten på avskjæringene

Jeg har testet de forskjellige avskjæringene som er beskrevet i avsnitt ´refkonkravs. Det gjorde jeg ved å kjøre programmet med og uten forskjellige avskjæringer. De forskjellige avskjæringene som er implementert er:

- Kun en transistor i hver posisjon
- Diffusjon utenfor rutingarealet
- Kollisjoner med nabopunktene
- Forbindelser til nabopunktene

Den første avskjæringen er så grunnleggende at den ikke kan fjernes uten at resten av programmet gir meningsløse resultater.

program	transistorer	plassering	løsninger	tidsforbruk
c_all	1	64	32	0.0
c_illegal	1	64	32	0.0
c_diff	1	64	32	0.0
c_bare	1	64	64	0.0
c_all	2	2112	640	0.1
c_illegal	2	2112	1024	0.0
c_diff	2	2112	1024	0.0
c_bare	2	4160	4096	0.0
c_all	3	43072	7680	1.3
c_illegal	3	67648	32768	1.1
c_diff	3	67648	32768	0.4
c_bare	3	266304	262144	1.4
c_all	4	534592	48240	17.6
c_illegal	4	2164800	1048576	46.5
c_diff	4	2164800	1048576	17.6
c_bare	4	17043520	16777216	114.1
c_all	5	3621952	107520	96.4
c_illegal	5	69273664	33554432	1502.8
c_diff	5	69273664	33554432	592.3
c_bare	5	*	*	3620.5

Tabell 4.2: Resultater av kjøring av Cell med forskjellige avskjæringer på utleggsareal med størrelse 5×5 punkter. * betyr at vi ikke fikk noen resultater fordi maskinens virtuelle minne ble fullt.

Men de tre neste kan fjernes, uten at det vil påvirke annet enn antall løsninger som blir funnet og tiden det tar.

Jeg kompilerte derfor spesialversjoner av Cell der de forskjellige avskjæringene var sjaltet inn/ut. For ikke å drukne i forskjellige kombinasjoner tester jeg ikke alle avskjæringene separat. Hvis jeg tar med den siste avskjæringen, tar jeg med alle. Og tar jeg med 'Kollisjoner med nabopunktene' tar jeg også med 'Diffusjon utenfor rutingarealet'.

For å få realistiske data, kjørte jeg partisjoneringsalgoritmen min på `ti_alu` (beskrevet i avsnitt 3.8.6). Jeg valgte ut den første gruppen som ble generert og brukte den som inndata til Cell. Dette simulerer dermed det som vil skje i et eventuelt ferdig system.

Disse programmene ble kjørt på en Sun-Sparcserver20, med resultat som vist i tabellene 4.1, 4.2 og 4.3.

Som vi ser er det generelt stor forskjell med og uten avskjæringer. Det er en av avskjæringene som peker seg ut som lite fordelaktig. Det er illegal som tester mot kollisjoner med nabopunktene. Det vil ikke forekomme noen tilfeller hvor denne avskjæringen har noen effekt, så det eneste den bidrar med, er å øke tidsforbruket.

I utgangspunktet kan det derfor virke som om dette er en svært uheldig avskjæring, men dette har sin forklaring. Dette er prisen man må betale for å la programmet være teknologiuavhengig, for dette er en avskjæring som kan få mye å si med andre teknologiregler. Også hvis arealet som transistorene skal plasseres på, har mange forhåndplasserte elementer, vil

program	transistorer	plassering	løsninger	tidsforbruk
c_all	1	100	60	0.2
c_illegal	1	100	60	0.2
c_diff	1	100	60	0.2
c_bare	1	100	100	0.2
c_all	2	6100	2744	0.3
c_illegal	2	6100	3600	0.1
c_diff	2	6100	3600	0.0
c_bare	2	10100	10000	0.0
c_all	3	280500	94536	11.7
c_illegal	3	366100	216000	6.8
c_diff	3	366100	216000	4.0
c_bare	3	1010100	1000000	5.1
c_all	4	9734100	2401224	364.5
c_illegal	4	21966100	12960000	414.5
c_diff	4	21966100	12960000	146.8
c_bare	4	101010100	100000000	525.5

Tabell 4.3: Resultater av kjøring av Cell med forskjellige avskjæringer på utleggsareal med størrelse 6×6 punkter.

denne avskjæringen kunne ha en gunstig effekt. Men til tross for dette kan det tenkes at den burde fjernes, siden den neste sjekken vil utføre de samme testene. Dette må undersøkes nærmere, med forskjellige teknologiregler.

Vi ser også at det er en av kjøringene som fikk spesielle problemer etter litt over en times kjøring:

```
c_bare 5 5 5: Virtual memory exceeded in `new'
  3620.5 real      3147.9 user        44.6 sys
```

Ved å koble ut alle avskjæringene (bortsett fra den helt nødvendige), ble minnekravet til prosessen større enn det virtuelle minnet på maskinen det ble kjørt på. Denne hadde 54 MB virtuelt minne, så minneforbruket til prosessen var betydelig. Heldigvis blir resultatet et helt annet med alle avskjæringene innkoblet: Denne versjonen finner over 100000 løsninger på litt over halvannet minutt. Dette viser at avskjæringene (med unntak av illegal) har stor effekt.

Egentlig burde jeg her ha kjørt Mighty for hver av løsningene for å evaluere tiden det tar før den første ekte løsningen blir funnet. Problemet er at Mighty i mange tilfeller henger seg opp (som nevnt på side 3.6.4). Derfor ble dette ikke gjort.

4.5 Ruting utført på samme måte som transistor-plasseringen

Arisland tenkte seg å bruke samme prinsipp på rutingen som for plassering av transistorene, dvs å prøve alle muligheter og å foreta avskjæringer. Han hadde begynt arbeidet med å utvide Cell for å utføre ruting på den måten. Dette stoppet opp fordi programmet ble vanskelig å

vedlikeholde, og fordi problemet er vanskelig i seg selv. Det første jeg gjorde var å skrive om Cell fra grunnen av, og gi det en god, objekt-orientert struktur som skulle være enkel å vedlikeholde og utvide. Deretter forsøkte jeg meg på rutingen, etter samme prinsipp som Arisland.

Det er relativt enkelt, spesielt ved bruk av `try_next()`, å lage en optimal ruter som bruker backtracking. Vanskeligheten er å finne gode nok avskjæringer til at programmet finner løsninger i rimelig tid. Etter å ha forsøkt mange forskjellige avskjæringer, fant jeg ut at det ville være fordelaktig å ha en fungerende ruter å sammenligne med. Derfor integrerte jeg Mighty inn som ruter, slik jeg har beskrevet i avsnitt 3.6.3. Med erfaringene fra en annen ruter, kikket jeg igjen på muligheten av å skrive en ruter som opererte ut i fra de samme prinsipper som Cell forøvrig.

Jeg har nå kommet til at årsaken til problemene med å finne gode nok avskjæringer, er at det rett og slett er håpløst mange muligheter.

Jeg stiller meg derfor tvilende til om dette er en god angrepsvinkel for å løse rutingproblemet. Det blir veldig mange muligheter å gå igjennom, mens hvis man legger litt mer struktur på problemet, kan man kanskje finne en løsning raskere.

4.6 Objekt-orientering. Fungerte det?

Dette er ikke en viktig del av oppgaven, men det er allikevel verdt noen ord. Det ble brukt mye tid på å finne en god struktur på programmet slik at det skulle være lett å skrive mer. Som vi så var det etter dette forarbeidet enkelt å lage implementasjoner for både plasseringen av transistorer i cell og løsningen av dronningproblemet. Konklusjonen her er at objekt-orientering ga oss et abstraksjonsnivå som gjorde deler av arbeidet enklere. Vedlikehold av programmet ble også enklere ved å innføre objekt-orientering.

4.7 Valget av InterViews

InterViews fungerte greit i min lille applikasjon, og det lot seg greit innordne i resten av min objektorienterte tankegang med gjenbruk av kode som et av målene. Et problem var at det er forskjellige versjoner av biblioteket, og de fungerte ikke sammen med alle kompilatorene. Men hvis man holdt seg til en versjon som fungerte, var det forsåvidt greit.

Dessverre er det ikke lenger så mange ute i den store verden som bruker det, og det foregår ikke lenger utvikling av det. Foreløpig finnes det ikke noe bibliotek, som ville vært aktuelt å bruke for meg, som fullt ut har den samme funksjonalitet som InterViews og som samtidig har mulighet for objekt-orientering.

Kapittel 5

Konklusjon

5.1 Bakgrunn

Jeg har i denne oppgaven sett på problemet med å lage et program for å legge ut en nettliste på så lite areal som mulig.

Rammen for oppgaven er Arislands ideer om hvordan man kan angripe dette problemet på en ny måte. Ved å splitte nettlisten opp i tilstrekkelig små deler, kan man lage et program som legger ut de små nettlistene optimalt til en minimodul.

Egentlig skulle jeg bare se på hvordan oppsplittingen kunne gjøres. Men jeg ble også trukket inn i arbeidet med å lage programmet for å legge ut de små nettlistene optimalt.

5.2 Programmet Cell

Jeg har laget et program, Cell, for å søke igjennom alle lovlige plasseringer av transistorene inne i minimodulene. Programmer bruker backtracking kombinert med avskjæringer for å søke igjennom mange muligheter på kort tid. Jeg har analysert avskjæringene for å finne ut hvor godt de fungerer.

Programmet er bygget på et tidligere program av Arisland, men har fått en bedre struktur. Mitt program er skrevet i en objekt-orientert stil som er enklere å vedlikeholde, og å utvide, enn det tilfellet var med det tidligere programmet. Blant annet har jeg laget en generell mekanisme for å skrive programmer som bruker backtracking. Ved å implementere et program som løser dronningproblemet, har jeg demonstrert at dette fungerer.

Programmet er teknologiuavhengig, slik at det skal være mulig å lage et slags manuelt ekspertsystem ved at desingeksperter kan legge inne spesialskrevne teknologiregler. Teknologireglene vil ha mye å si for hvor godt resultat dette vil gi, siden de vil påvirke oppløsningen på rutemøsnteret som brukes.

Det er implementert et grafisk grensesnitt til Cell (og til dronningprogrammet) slik at dataene kan bli visualisert.

5.3 Ruting i minimodulene

Jeg har forsøkt å lage en ruter etter samme oppskrift som ved plasseringen av transistorene uten positivt resultat. Jeg tror dette kommer av at det rett og slett er for mange mulige løsninger til at denne strukturen er velegnet. Selv med kraftige avskjæringer, vil det være for mange posisjoner å studere.

For å få erfaringer med en ruter, har jeg koblet Cell sammen med Mighty, som er en anerkjent koblingsboksruter. Dette gjør at Cell fungerer som et utleggsprogram. Jeg har funnet noen svakheter med Mighty som tilsier at den ikke bør brukes i den endelige versjonen, men stort sett fungerer Mighty greit. Resultatet av rutingen blir også inkorporert i Cell, og blir vist frem i det grafiske grensesnittet.

5.4 Partisjonering

Etter å ha gjort meg ferdig med Cell, returnerte jeg til det jeg hadde tenkt til å gjøre fra starten av, nemlig oppgaven med å splitte opp nettlisten i mindre deler. På dette området er det en omfattende litteratur. Det var en stor oppgave å systematisere denne informasjonen.

Et inntrykk jeg fikk etter å ha studert denne litteraturen, var at det er gjort forholdsmessig mye for å finne gode algoritmer, mens arbeidet med å finne gode kriterier ikke har kommet like langt. Dette er litt underlig, siden det å finne det riktige kriteriet kan være viktigere enn å finne en god algoritme, som løser et annet problem enn det man egentlig ønsker.

De kriteriene som er foreslått har nok ofte sin plass, men for meg som skal bruke dette til å splitte opp en nettliste basert på transistorer i CMOS teknologi, har de store svakheter. Jeg tenker her f.eks på det faktum at man i CMOS har to typer transistorer, som bør havne i forskjellige grupper for at utlegget skal bli kompakt.

Jeg har også studert grensesnittet mellom partisjonering og plassering. Jeg fant at den oppdelingen i faser som har vært vanlig, gjør at partisjoneringsalgoritmen har litt for lite informasjon om plasseringen til at den kan gjøre gode valg. Derfor har jeg foreslått en endring i grensesnittet mellom de to fasene, slik at det blir sendt over litt mer informasjon. Ved at partisjoneringsalgoritmen får anledning til å angi naboforhold, vil man kunne oppnå resultater som er bedre egnet som utgangspunkt for et ferdig utlegg. Jeg vil fremheve dette som det viktigste bidraget i denne oppgaven, og jeg tror at dette kan være verdifullt også for andre forskere på dette feltet.

Tankene omkring kriterier kulminerer i et kriterium for k-partisjonering. For å få praktisk erfaring med bruken av kriteriet, har jeg implementert en algoritme som bruker det. Algoritmen er ikke så gjennomarbeidet som kriteriet, men den har vært verdifull for å teste ut nye ideer. Et annet hjelpemiddel til uttestingen har vært testdata som stammer fra realistiske kretser.

Jeg har også implementert Kernighan-Lin algoritmen, for å gjøre meg kjent med denne klassiske algoritmen og med testdataene.

5.5 Forskjellige poenger

I tillegg til det som er nevnt, har jeg kommet med forskjellige ideer til totalverktøyet. Jeg har beskrevet hvordan man kan la verktøyet oppfylle eksterne krav til høyde/bredde-forhold på modulene.

Jeg har også beskrevet hvordan man kan utnytte parallellitet i implementasjonen av verktøyet.

5.6 Om nye ideer

Ved å studere det som er gjort tidligere, kan man ofte kombinere metoder, og raffinere algoritmer. På den måten kan man få resultater som er litt bedre enn tidligere. I Arislands arbeid og i denne oppgaven er det fulgt en annen strategi. Det å prøve med en helt ny angrepsvinkel til problemet, er som å lete etter en snarvei. Hvis man finner den, kan man tjene mye. Hvis man ikke finner den, bruker man lenger tid, kanskje mye lenger tid. Selv om man ikke finner den, kan man likevel ha fått ny kunnskap om landskapet rundt, og kanskje ideer til nye snarveier.

Om "snarveien" i denne oppgaven fører fram, er det enda for tidlig å si noe om. Siden det er originalt arbeid, er det vanskelig sammenligne det med tidligere algoritmer før alt er ferdig. Men jeg har ihvertfall fått ny innsikt i problemet, og jeg håper at denne innsikten vil være nyttig også for andre.

En observasjon på veien, er at mange er fornøyd med å følge andres fotspor, selv om de kanskje ikke skulle samme vei. Det er få som analyserer problemene inngående, for å finne trekk ved de konkrete problemene som er viktige for å få gode resultater.

Kapittel 6

Videre arbeid

6.1 Hele verktøyet

Denne oppgaven har bare vært ment som en del av et større arbeid. Derfor gjenstår det mye arbeid før hele verktøyet er klar til å testes i sin helhet. Det mest interessante spørsmålet er om strukturen av modulgeneratoren er egnet til formålet. Vil man kunne oppnå resultater som kan konkurrere med det beste andre har gjort? Hvis det viser seg at metoden er bra, vil det være aktuelt å gjøre programmene egnet til praktisk bruk, ved at de leser og skriver filer i standard formater, osv.

Underveis til dette målet, er det flere delmål. Ikke minst å gjøre programmene så ferdige at de kan testes. Dette vil kunne gi verdifulle hint til hva som er riktige valg, og vil dermed være med på å styre den videre utviklingen.

6.2 Partisjonering

De foreløpige erfaringene med å inkludere naboskap som en del av kriteriet for partisjoneringen virker lovende. Dette bør utvikles videre, for å se hvor mye det påvirker arealet på ferdige utlegg. For å få full effekt av dette, må plasseringsprogrammet også ta hensyn til naboer.

6.3 Plassering av transistorer

Programmet som plasserer transistorer er i hovedsak ferdig. Men jakten på nye avskjæringer vil fremdeles være viktig for å oppnå god ytelse.

Teknologireglene kan være en effektiv mekanisme for å få Cell til å lage kompakte utlegg. Dette bør utvikles videre av noen med erfaring fra å designe kretser.

6.4 Ruting

Fordi Mighty har flere svakheter, bør man enten utbedre disse, eller (enda bedre) utvikle en egen ruter som er spesialskrevet til omgivelsen.

Generelt

Mine tanker omkring oppdeling i faser tror jeg kan brukes også i andre delproblemer i VLSI. Ved å studere grensesnittet mellom de forskjellige fasene, kan man muligens finne andre faser som kan integreres på nye og bedre måter.

Dette kunne det være interessant å studere nærmere, selv om det er vanskelig å si på forhånd hvilke resultater man kan forvente.

Referanser

Agrawal, P., & Brauer, M. A. (1977). Some theoretical aspects of algorithmic routing. In *Proc. 14th Design Automat. Conf.*, pp. 23–31.

Alpert, C. J., & Kahng, A. B. (1993). Geometric embedings for faster and better multi-way netlist partitioning. In *Proc. ACM/IEEE Design Automat. Conf.*, pp. 743–748.

Alpert, C. J., & Kahng, A. B. (1994a). A general framework for vertex orderings, with applications to netlist clustering. In *Proc. of IEEE Int. Conf. on CAD (ICCAD)*, pp. 63–67.

Alpert, C. J., & Kahng, A. B. (1994b). Multi-way partitioning via spacefilling curves and dynamic programming. In *Proc. ACM/IEEE Design Automat. Conf.*, pp. 652–657 San Diego, USA.

Arisland, K. Ø. (1989). Automatic VLSI layout synthesis. Unpublished.

Avis, D. (1983). A survey of heuristics for the weighted matching problem. *Networks*, *13*(3), 475–493.

Barnes, E. R. (1982). An algorithm for partitioning the nodes of a graph. *SIAM J. of Algebraic and Discrete Methods*, *3*(4), 541–550.

Breuer, M. A. (1977a). A class of min-cut placement algorithms. In *Proc. 14th Design Automat. Conf.*, pp. 284–290.

Breuer, M. A. (1977b). Min-cut placement. *J. Des. Autom. & Fault-Tolerant Comput.*, *1*(4), 343–362.

Bui, T. N., Chaudhuri, S., Leighton, F. T., & Sipser, M. (1987). Graph bisection algorithms with good average case behavior. *Combinatorica*, *7*(2), 171–191.

Bui, T. N., Heigham, C., Jones, C., & Leighton, T. (1989). Improving the performance of the Kernighan-Lin and simulated annealing graph bisection algorithms. In *Proc. 26th ACM/IEEE Design Automation Conf.*, pp. 775–778.

Bui, T. N., & Moon, B. R. (1994). A fast and stable hybrid genetic algorithm for the ratio-cut partitioning problem on hypergraphs. In *31st Design Automat. Conf.*, pp. 664–669.

Chan, P. K., Schlag, M. D. F., & Zien, J. Y. (1994). Spectral k-way ratio-cut partitioning and clustering. *IEEE Trans. on CAD*, *13*, 1088–1096.

Cohoon, J. P., & Heck, P. L. (1988). Beaver: a computational-geometry-based tool for switchbox routing. *IEEE Trans. on CAD*, *cad-7*(6), 684–697.

Cong, J., & Smith, M. (1993). A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design. In *30st Design Automat. Conf.*, pp. 755–760.

Domic, A., Levitin, S., Phillips, N., Thai, C., Shiple, T., Bhavsar, D., & Bissell, C. (1989). Cleo: a CMOS layout generator. In *Proc. Seventh IEEE Int. Conf. on Comp.-A. Des.: ICCAD-89*, pp. 340–343 Santa Clara, California, USA. Digest of Technical Papers.

Donath, W. E., & Hoffman, A. J. (1973). Lower bounds for the partitioning of graphs. *IBM J. Res. Dev.*, *17*, 420–425.

Doy, J., Maly, W., & Thomas, M. E. (1987). Detection and physical characterization of spot defects in metal interconections. In *172nd Electrochemical Society Meeting*, p. 837.

Dunlop, A. E., & Kernighan, B. W. (1985). A procedure for placement of standard-cell VLSI circuits. *IEEE Trans. on CAD*, *4*(1), 92–98.

Feo, T. A., & Khellaf, M. (1987). A class of bounded approximation algorithms for graph partitioning. typescript.

Fiduccia, C. M., & Mattheyses, R. M. (1982). A linear-time heuristic for improving network partitions. In *ACM IEEE 19. Design Automation Conf. Proc.*, pp. 175–181 Las Vegas.

Ford, L. R., & Fulkerson, D. R. (1962). *Flows in Networks*. Princeton Univerity Press, Princeton, New Jersey.

Galil, Z. (1986). Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys*, *18*(1), 23–38.

Garbers, J., Promel, H. J., & Steger, A. (1990). Finding clusters in VLSI circuits. In *Proc. of IEEE Int. Conf. on CAD (ICCAD)*, pp. 520–523.

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co.

Goldschmidt, O., & Hochbaum, D. S. (1994). A polynomial algorithm for the k-cut problem for fixed k. *Math. of Operations Research*, *19*(1).

Golomb, S. W., & Baumert, L. D. (1965). Backtrack programming. *Journal of the ACM*, *12*, 516–524.

Griffith, J., Robins, G., Salowe, J. S., & Zhang, T. (1994). Closing the gap: Near-optimal steiner trees in polynomial time. *IEEE Trans. on CAD*, *13*, 1351–1365.

Hagen, L., & Kahng, A. (1992a). New spectral methods for ratio cut partitioning and clustering. *IEEE Trans. on CAD*, *11*, 1074–1085.

Hagen, L., & Kahng, A. B. (1992b). A new approach to effective circuit clustering. In *1992 IEEE/ACM Int. Conf. on Computer-Aided Design. Digest of Technical Papers (Cat. No.92CH03183-1)*, pp. 422–427 Santa Clara, CA, USA.

Hall, K. M. (1970). An r-dimensional quadratic placement algorithm. *Managment Science*, *17*(3), 219–229.

Hérault, L., & Niez, J.-J. (1989). How neural networks can solve hard graph problems: a performance study on the graph k-partitioning. In *Neuro-Nimes' 89. Int. Works. Neural Networks and Their Application*, pp. 237–255.

Hertz, J., Krogh, A., & Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley.

Hoyle, R., Priest, A., & Hetherington, G. (1991). Alternative control strategies for rule-based transistor placement in CMOS VLSI design. *IEE Proceedings-G Circuits Devices and Systems*, *138*(1), 137–144.

Hsu, C.-P. (1984). Minimum-via topological routing. *IEEE Trans. on CAD*, *cad-3*(3), 184–190.

Hughes, T. A., Salama, R., & Liu, W. (1986). BBC: A module generator for back-to-back cells. In *Fourth IEEE Int. Conf. on Comp.-A. Des.: ICCAD-86*, pp. 440–443 Santa Clara, California, USA. Digest of Technical Papers.

Hyafil, L., & Rivest, R. L. (1973). Graph partitioning and constructing optimal decision trees are polynomial complete problems. Rep. 33, IRIA-Laboria. Vanskelig fysisk tilgjengelig, referert av (Dunlop & Kernighan, 1985; Garey & Johnson, 1979).

Johnson, D. S., Aragon, C. R., McGeoch, L. A., & Schevon, C. (1989). Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Oper. res. (USA)*, *37*(6), 865–892.

Kernighan, B. W., & Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, *49*(2), 291–307.

Kim, J., McDermott, J., & Siewiorek, D. (1984). Exploiting domain knowledge in IC cell layout. *IEEE Design and Test*, *1*(3), 52–64.

Kirkpatrick, S. (1984). Optimization by simulated annealing — quantitative studies. *J. of Statistical Physics*, *34*(5), 975–986.

Knudsen, T. (1991). Automatisk lokalruting av VLSI-utlegg. Hovedoppgave, Universitetet i Oslo.

Knuth, D. E. (1975). Estimating the efficiency of backtrack programs. *Mathematics of Computation*, *29*, 121–136.

Kollaritsch, P., Lusky, S., Prasad, S., & Potter, N. (1988). CLAY: a malleable-cell, multi-cell, transistor matrix approach for CMOS layout synthesis. In *Sixth IEEE Int. Conf. on Comp.-A. Des.: ICCAD-88*, pp. 142–145 Santa Clara, California, USA. Digest of Technical Papers.

Kollaritsch, P. W., & Weste, N. H. E. (1985). TOPOLOGIZER: An expert system translator of transistor connectivity to symbolic cell layout. *IEEE J. of Solid-State Circuits*, *20*(3), 799–804.

Krishnamurthy, B. (1984). An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, *C–33*(5), 438–446.

Krishnamurthy, B. (1987). Constructing test cases for partitioning heuristics. *IEEE Transactions on Computers*, *C-36*(9), 1112–1114.

Kuh, E. S., & Marek-Sadowska, M. (1986). *Layout Design and Verification*, chap. Global Routing, pp. 169–197. Elsevier Science Publishers B.V. (North-Holland).

Kuh, E. S., & Ohtsuki, T. (1990). Recent advances in VLSI layout. *Proc. of IEEE*, *78*(2), 237–263.

LaPaugh, A. S., & Pinter, R. Y. (1989). Channel routing for integrated circuits. *Annu. Rev. Comp. Sci.*, *4*, 307–363.

Laszewsski, G., & Mühlenbein, H. (1991). Partitioning a graph with a parallel genetic algorithm. In Schwefel, H., & Manner, R. (Eds.), *Parallel Problem Solving from Nature*, pp. 165–169. Springer-Verlag.

Lee, C. Y. (1961). An algorithm for path connections and its applications. *IRE Trans. Electron. Comput.*, *EC-10*(3), 346–365.

Lee, J., Chou, J.-H., & Fu, S.-L. (1993). Clustering algorithm for network partitioning with a hypergraph model. *J. of Information Science and Engineering*, *9*(3), 359–378.

Leighton, F. T., & Rao, S. (1988). An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th IEEE Symp. on the Foundations of Computer Science (FOCS)*, pp. 422–431.

Leiserson, C. E. (1980). Area-efficient graph layout (for VLSI).. In *Proc. 21th Annual Symp. on the Foundations of Computer Science*, pp. 270–281 New York. IEEE.

Lengauer, T. (1990). *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons Ltd.

Liu, J. W. H. (1989). A graph partitioning algorithm by node separators. *ACM Transactions on Mathematical Software*, *15*(3), 198–219.

Lundh, Y., Søråsen, O., Bayegan, M., & Pedersen, J. E. (1983). *Konstruksjon av Integrerte Kretser*. Universitetsforlaget.

Madsen, J. (1989). A new approach to optimal cell synthesis. In *Seventh IEEE Int. Conf. on Comp.-A. Des.: ICCAD-89*, pp. 336–339 Santa Clara, California, USA. Digest of Technical Papers.

Mayrhofer, S., & Lauther, U. (1990). Congestion-driven placement using a new multi-partitioning heuristic. In *1990 IEEE Int. Conf. on Computer-Aided Design. Digest of Technical Papers (Cat. No.90CH2924-9)*, pp. 332–335.

Meuenier, F. (1989). A massivly parallelizable heuristic method for the graph partitioning problem. In *High Performance Computer Proc. of the Int. Symp.*, pp. 109–118.

Moore, E. F. (1959). Shortest path through a maze. *Annals of the Computational Laboratory of Harvard University*, *30*, 285–292.

Næss, S. (1991). Automatisk globalruting av VLSI-utlegg. Hovedoppgave, Universitetet i Oslo.

Namekawa, T., Suzuki, K., Takano, H., & Ohtsuki, T. Promoting efficiency of rip-up and reroute method and its evaluation (in japanese). IEICE monograph, VLD 88-91, pp. 39–46, 1989.

Poirer, C. (1987). EXCELLERATOR: automatic leaf cell layout agent. In *Fifth IEEE Int. Conf. on Comp.-A. Des.: ICCAD-87*, pp. 176–179 Santa Clara, California, USA. Digest of Technical Papers.

Rao, D. S., & Patnaik, L. M. (1989). Neural network based approach to standard cell placement. *Electronics Letters*, *25*(3), 208–209.

Riess, B., Doll, K., & Johannes, F. (1994). Partitioning very large circuits using analytical placement techniques. In *31st Design Automat. Conf.*, pp. 646–651.

Riezenman, M. (1991). Chips that work. *Byte*, *16*(8), 187–190.

Sanchis, L. A. (1989). Multiple-way network partitioning. *IEEE Transactions on Computers*, *38*(1), 62–81.

Saran, H., & Vazirani, V. (1995). Finding k cuts within twice the optimal. *SIAM J. Comput.*, *24*(1), 101–108.

Savage, J. E., & Wloka, M. G. (1991). Parallelism in graph-partitioning. *J. of Parallel and Distributed Computing*, *13*(3), 257–272.

Schweikert, D. G., & Kernighan, B. W. (1972). A proper model for partitioning of electrical circuits. In *Proc. Design Automation Workshop*, pp. 56–62.

Shin, H., & Kim, C. (1993). A simple yet effective technique for partitioning. *IEEE Trans. on VLSI Systems*, *1*(3), 52–64.

Shin, H., & Sangiovanni-Vincentelli, A. (1986). Mighty: A rip-up and reroute detailed router. In *Proc. of IEEE Int. Conf. on CAD (ICCAD)*, pp. 2–5.

Shin, H., & Sangiovanni-Vincentelli, A. (1987). A detailed router based on incremental routing modifications: Mighty. *IEEE Trans. on CAD*, *cad-6*(6).

Shing, M. T., & Hu, T. C. (1986). A decomposition algorithm for multi-terminal network flows. *Discrete Applied Mathematics*, *13*(2), 165–181.

Stroustrup, B. (1991). *The C++ Programming Language* (Second edition). Addison-Wesley.

Stroustrup, B., & Ellis, M. A. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley.

Suaris, P. R., & Gershon, K. (1989). A quadrisection-based combined place and route scheme for standard cells. *IEEE Trans. on CAD*, *cad-8*(3), 234–244.

Sun, P. K. (1989). Cetus — a versatile custom cell synthesizer. In *Seventh IEEE Int. Conf. on Comp.-A. Des.: ICCAD-89*, pp. 348–351 Santa Clara, California, USA. Digest of Technical Papers.

Szymanski, T. G. (1985). Dogleg channel routing is NP-complete. *IEEE Trans. on CAD*, *cad-4*(1), 31–41.

Talbi, E.-G., & Bessière, P. (1991). A parallel genetic algorithm for the graph partitioning problem. In *Proc. of the Int. Conf. on Supercomputing* Cologne.

Wei, Y.-C., & Cheng, C.-K. (1989). Toward efficient hierarchical designs by ratio cut partitioning. In *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 298–301.

Wei, Y.-C., & Cheng, C.-K. (1991). Ratio cut partitioning for hierarchical designs. *IEEE Trans. on CAD*, *10*, 911–921.

Weste, N., & Eshraghian, K. (1985). *Principles of CMOS VLSI Design*. Addison-Wesley.

Yeh, C.-W., Cheng, C.-K., & Lin, T.-T. Y. (1992). A probabilistic multicommodity-flow solution to circuit clustering problems. In *1992 IEEE/ACM Int. Conf. on Computer-Aided Design. Digest of Technical Papers (Cat. No.92CH03183-1)*, pp. 428–431 Santa Clara, CA, USA.